TAMPEREEN TEKNILLINEN YLIOPISTO

Hänninen, Arttu

Enterprise Integration Patterns in Service Oriented Systems

Master of Science Thesis

Examiner: Prof. Tommi Mikkonen

Examiners and topic approved in
the council meeting of Faculty of
Information Technology on
April 3rd, 2013.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
Hänninen, Arttu: Enterprise Integration Patterns in Service Oriented Systems
Diplomityö, 58 sivua
Kesäkuu 2014
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Tommi Mikkonen
Avainsanat: Enterprise Integration Patterns, Palvelukeskeinen arkkitehtuuri (SOA), Viestipohjainen integraatio

Palvelupohjaisen integraation toteuttaminen mihin tahansa tietojärjestelmään on haastavaa, sillä integraatioon liittyvät järjestelmät voivat muuttua jatkuvasti. Integraatiototeutusten tulee olla tarpeeksi joustavia, jotta ne pystyvät mukautumaan mahdollisiin muutoksiin. Toteutukseen voidaan käyttää apuna eri sovelluskehyksiä, mutta ne eivät välttämättä takaa mitään standardoitua tapaa tehdä integraatio. Tätä varten on luotu joukko ohjeita (Enterprise Integration Patterns, EIP), jotka kuvaavat hyväksi havaittuja tapoja tehdä integraatioita. Tässä työssä keskitytään näiden mallien tutkimiseen ja siihen, miten niitä voidaan hyödyntää yritysjärjestelmissä. Jotta tutkimukseen saadaan konkreettinen vertailutulos, erään järjestelmän integraatioratkaisu tullaan päivittämään uuteen. Uusi ratkaisu hyödyntää sovelluskehystä, joka perustuu EIP:eihin.

Työ jakautuu kolmeen osioon. Ensimmäisessä osassa selvitetään mitä palvelupohjainen integraatio on ja miten se on kehittynyt ajan kuluessa. Tässä osassa esitellään myös eri sovelluskehyksiä, joita voidaan hyödyntää integraation rakentamisessa. Toisessa osassa esitellään järjestelmä (Valtimo), jonka integraatioratkaisu tullaan päivittämään. Tässä osassa etsitään myös kriteereitä, joiden perusteella voidaan valita paras työkalu integraation rakentamiseen. Kriteerit ovat rajoitettu niin, että vain järjestelmään oleellisesti liittyvät kriteerit otetaan arviointiin mukaan ja nämä painotetaan vielä erikseen. Kolmas osa esittelee miten varsinainen toteutus suoritettiin edellisessä osiossa valitulla työkalulla.

Tutkimuksessa selvisi, että EIP:ien käyttö voi auttaa integraation rakentamista takaamalla tietyn joukon ominaisuuksia, joita sovelluskehysten on tarjottava. EIP:it pyrkivät samalla ohjeistamaan integraation rakentamista mahdollisimman järkeväksi. Toteutuksessa jää kuitenkin paljon itse sovelluskehyksen toteutuksen varaan. Oikean työkalun löytäminen integraatioon on erittäin tärkeää ja sitä varten tulisi työkaluja aina vertailla käyttötarkoituksen mukaisesti valittuja kriteereitä vastaan. Kriteerien tulee olla aina valittu niin, että ne ovat merkittäviä järjestelmälle, johon integraatio luodaan. Valtimon kohdalla uudella integraatioratkaisulla huomattiin joitain eroja vanhaan. Uusi ratkaisu tarjoaa laajemman joukon ominaisuuksia, joten ratkaisusta tuli joustavampi. Lisäksi käytetyllä työkalulla on aktiivisempi yhteisö, jonka johdosta sillä on luultavasti myös pidempi elinaika ja tiheämpi päivitysväli.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
Hänninen, Arttu: Enterprise Integration Patterns in Service Oriented Systems
Master of Science Thesis, 58 pages
June 2014
Major: Software Engineering
Examiners: Prof. Tommi Mikkonen
Keywords: Enterprise Integration Patterns, Service Oriented Architecture, Messaging, Integration

Enterprise Integration is difficult to implement, since the environments around it are constantly changing. Some tools and frameworks can help the implementation, but they might not have any standardized way of creating the integration. Enterprise Integration Patterns will help with this by giving a set of patterns as guidelines on how the integration should work. This thesis examines what these patterns are and how exactly they impact the integration process. To get some concrete results, one integration solution will be improved by using a tool that is based on EIPs.

The thesis is divided into three parts. The first part examines the theory behind Enterprise Integration. It explains the evolution of Enterprise Integration and introduces some integration frameworks that can help the implementation. The second part revolves around Valtimo, the application that will receive the improved integration solution. In it, the criteria for choosing the best tool are examined. From the criteria, seven are chosen as the most relevant for Valtimo, and all the tools are evaluated against these weighed criteria. The final part explains how the actual new implementation was made with the chosen tool.

The study indicates that Enterprise Integration Patterns can help the integration by guaranteeing some features to be available in a framework, and bringing some best practices to the implementation. Still, a lot is dependant on how the actual integration tool is implemented. To choose the right tool, some set of criteria should be always applied. The criteria should be chosen so that there are only relevant ones to the current case. Comparing the new solution in Valtimo to the old one, some noticeable differences were realized. The new solution has a richer set of features, more active community and thus longer lifespan, and the tool is being kept up to date more frequently.

PREFACE

This thesis was made for the Master of Science degree in software engineering at Tampere University of Technology. It was made in cooperation with Gofore Oy, and I would like to thank them for giving me an opportunity to work with an interesting and challenging subject.

I would also like to thank my examiners professor Tommi Mikkonen, Jarkko Hyöty, and Tapio Rautonen for their valuable input and guidance. With their help I was able to make the thesis much more coherent and better structured. As for the integration implementation, I would like to thank my co-workers, especially Marko Hollanti and Janne Mattila, for helping me with the solution whenever needed. Lastly, I would like to thank my family and friends for their support, and keeping me motivated to finish the work.

Tampere, May 16, 2014

Hänninen, Arttu

CONTENTS

ABBREVIATIONS

API	Application Programming Interface
BPEL	Business Process Execution Language
CSV	Comma-separated Values
EIP	Enterprise Integration Patterns
EJB	Enterprise JavaBeans
ESB	Enterprise Service Bus
DSL	Domain Specific Language
IDE	Integrated Development Environment
JAXB	Java Architecture for XML Binding
JBI	Java Business Integration
JMS	Java Message Service
JPA	Java Persistence API
JSF	JavaServer Faces
MQ	Message Queue
POM	Project Object Model
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
UI	User Interface
WSDL	Web Services Description Language
XML	Extensive Markup Language
XSL	Extensible Stylesheet Language

1. INTRODUCTION

In today's world, everything is connected. That also usually means that applications can rarely live in isolation. Whether it is an e-commerce service checking inventory, or a healthcare service looking up patient information, integration is essential in any larger enterprise application. Creating a single large application to run a complete business is from hard to impossible. Integration can provide flexibility to the system and let the business choose best parts for their needs. The applications can be built in house or by third-party vendors, and they are usually run on multiple computers. [1, p. 1-2]

Enterprise Integration means connecting systems, companies, and/or people. The definition is quite wide and can include many things. Usually the integration is about two or more systems sharing data or making requests to create, remove or change data. [1, p. 6-8]

Implementing an integration solution is never easy. There are a number of challenges that developers face. Networks are unreliable or slow, so that data may be lost between the systems or take too long to get to their destination. Applications can use a wide variety of technologies and programming languages, yet the integration solution should be able to transfer data between them. The applications are also constantly changing, so the integration solutions should have minimal dependency on the applications. [1]

Over time these problems have been overcome with different patterns and development models. These patterns have been evolving to the point where Gregory Hohpe and Bobby Woolf finally put up a list of currently 65 Enterprise Integration Patterns (EIP) [2]. This list is bound to change as time passes and more patterns are discovered and some patterns found to be non-optimal. The patterns are based on messaging, and aim to help developers implement integration solutions as efficiently as possible. They also try to keep the architecture as simple and understandable as possible. Without proper planning, integration architecture can become overly complicated and impossible to manage.

In this thesis, an integration solution will be implemented to Valtimo, a software to

manage the processes involved in the occupational safety control of Finland. The project currently has a working integration solution, but it can be improved. The purpose of it is to connect a web UI and a WSDL interface to a service layer. The web UI and the service layer are located in different environments. The problem with the old solution is that it is built with a product that uses an older, discontinued specification. This causes some uncertainty for the future of the product. Combined with the need to extend the integration solution further, a decision was made to look for a better solution. The integration is service based, thus Service-Oriented Architecture is closely related to the solution. Since the integration solution has also a WSDL interface, understanding Web Services is also necessary. Valtimo uses SOAP as the Web Service implementation method.

This thesis is structured so that it first describes the theoretical aspect of integration in Chapter 2. It mainly consists of the evolution of integration solutions and the variety of frameworks available to help the implementation. Chapter 3 introduces project Valtimo on a deeper level. It also describes the criteria, that can be used when choosing the right integration tool for the implementation. There is also a big question of whether to use an Enterprise Service Bus (ESB) or not. From the criteria, an evaluation model will be created, which grades the different integration tools and points us to the best tool for case Valtimo. With this tool, an integration solution will be implemented and described in Chapter 4. It also includes an overview of the integration architecture by using the Enterprise Integration Patterns. Chapter 5 concludes the thesis by comparing the new solution to the earlier one and having some final words on how the EIPs can help the integration process.

2. ENTERPRISE INTEGRATION

This chapter describes how integration is done in enterprise applications. Enterprise integration as a field has been around for a long time. It aims to study communication and data exchange between multiple systems. Since the needs for communication are unpredictable and continuously changing, enterprise integration focuses on optimizing operations in that sense. However, all integration solutions face some common challenges. One problem comes from the network, which can be unreliable and slow, causing data to sometimes be either lost in the way or significantly delayed. Another problem is different environments. Data needs to be transmitted between completely different systems with possibly their own programming languages, platforms and data types. Also, integration solutions need to minimize dependencies between systems.

Integration is a wide concept, and it can happen on many levels. In a simple case, it might mean the process of finding information from multiple locations and showing the result to the user. Another case could be multiple systems using the same data or functionality. In the former case, data would need to be replicated between systems and kept up to date. In the latter case, some functionality may need to be shared between multiple applications. This concept can be expanded into Service-Oriented Architecture, where the functions are replaced with the concept of services. [1, p. 6-8]

2.1. Service-Oriented Architecture

Designing enterprise applications has become increasingly more difficult since the revolution of internet and information economy. Everything needs to be connected and accessible at all times. This also means that applications are constantly getting bigger and more complex. To be able to meet all the necessary requirements of the always changing environments, the applications must be designed in a way that provides efficiency, flexibility and adaptability. Service-oriented architecture (SOA) solves these problems by focusing

on these aspects. [3]

The idea of SOA is to form collections of independent software modules (services), that can be combined to form a larger system. The services can be reused in other systems, thus bringing flexibility, adaptability and reusability for the architecture. These services are also loosely coupled, meaning that they work autonomically, without knowing much of any other services. The autonomy of the services brings also a layer of abstraction to the system. It hides details of how the services are built and instead presents some pre-built components with specific interfaces and usages. A simple case where a service consumer requires a task from a service provider can be seen in Figure 2.1. [4]

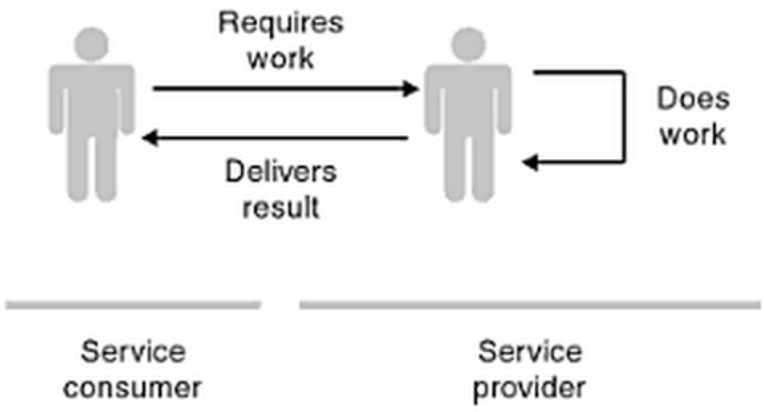

Figure 2.1: The concept of service-orientation. Service consumers (client) use the services from service providers (server). [3]

SOA is also not just an architecture from technical perspective. It consists of set of rules, policies, and practises to ensure the quality of the products and that the right services are used. SOA Manifesto [5] has the following priorities for service orientation: business value over technical strategy, strategic goals over project-specific benefits, intrinsic interoperability over custom integration, shared services over specific-purpose implementations, flexibility over optimization, and evolutionary refinement over pursuit of initial perfection. These principles favour heavily organizational goals over technical goals. Thus implementing SOA on organizational level may require some fundamental changes inside the organization.

The main advantages of service-oriented approach are agility and cost-efficiency. Agility here means that SOA enables quick response times to the fast-changing market and customer needs due to the nature of services. Cost-efficiency comes from the

reusability of the services. While the advantages are many, there are still some disadvantages too. For one, designing and managing a service-oriented system can be complex and time-consuming. There are usually a lot of services in different environments, which produce a lot of messages. For smaller applications, implementing SOA may bring a lot of unnecessary complexity and only little advantages. Another challenges include incomplete standards, corporate policies (how well SOA is received on a corporate level) and security in SOA. [6]

SOA is not tied to any specific technology or platform, meaning that the services can have multiple interfaces for different means of communication [6]. This brings interoperability to SOA and allows different systems to communicate with each other. As for designing enterprise software, SOA meets a lot of the common requirements. These include simplicity of the architecture (in order to enhance efficiency), flexibility and maintainability to take changes in to account, reusability to avoid redundancy, and finally decoupling to make the enterprise independent of specific technologies. [3]

The concept of services and service-orientation has been around for a long time, and it has had many different definitions. In fact, it can be said that SOA is not a new concept at all, but instead evolution of existing processes. In the recent years, SOA has been rising in popularity quite heavily and more often linked to Web Services [3]. Still, while SOA has been a long time coming, it is not as popular yet as expected. This is because the transition to service orientation requires the support from people, policies and practices on organizational and industrial level. Organizations, especially big ones, will not quickly adapt to anything new. [6]

2.2. Web Services

Defining the term "Web services" is not easy, since there are many definitions of it on different abstraction levels. On an abstract level, it is communication between two devices over the World Wide Web (WWW). W3C defines it as a software system, which is designed to support interoperable machine-to-machine interaction over a network [7]. Going to a more specific level, there are many types of Web Services. The most common ones are SOAP (originally defined as Simple Object Access Protocol) and Representational State Transfer (REST) [8]. To actually implement a system using Web Services,

some kind of agent is required to send and receive messages.

SOAP is a protocol that relies on using structured information to transfer messages over a network. The messaging is XML-based and the messages are transferred usually by HTTP or SMTP. They consist of three parts: envelope for telling what is in the message and how it will be processed, encoding rules for application-defined data types, and a way to represent procedure calls. SOAP can also handle security and transaction issues with some extensions. [7]

A picture of Web Services architecture using SOAP can be seen in Figure 2.2. The architecture shares a lot of similarities to SOA architecture. That is because Web services are a realization of SOA. Service requester is a person or organization that requires some service from the service provider. Service provider in turn is a person or organization providing an agent to implement that service. The rules of communication, if necessary, can be defined with Web Services Description Language (WSDL). [7]

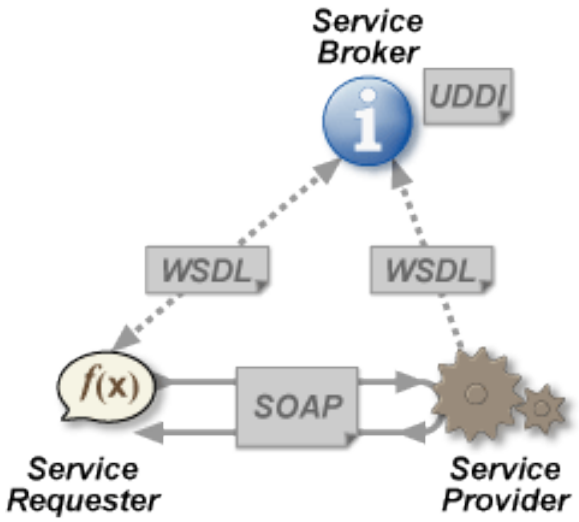

Figure 2.2: Web Services architecture using SOAP and WSDL. [9]

REST, on the other hand, describes a set of principles, or architectural constraints by which the data can be transmitted. It is a lot simpler and lighter solution that SOAP. It does not contain any additional layers, and is focused to create stateless services. To access the resources, an unique identifier (URI) is used and manipulated through four basic operations: GET, POST, PUT, DELETE. When accessing the information, it can be in any format, such as JSON or XML. It is left to the developers to handle everything else than the actual transportation of the message. [8]

Comparing the two, REST is lightweight and loosely coupled, leaving it to the developer to handle everything as he/she sees fit. SOAP uses strictly defined format to force the messages to be structured in a specific way. This introduces some restrictions to messaging, but also helps developers to know exactly what kind of messages are coming in. SOAP also adds some layers, such as transaction and security, to help the development. [8]

The most significant advantage of using Web services is increased interoperability. Implementations do not pose any requirements to platform or programming language, and use standardized communication methods. Another great advantage is, that like SOA, the implementations also have a strong reusability factor due to the nature of services. That also leads to cost-efficiency. [7]

2.3. Integration Solutions

To achieve integration between multiple systems, some kind of solution needs to be implemented. There are a lot of tools and technologies to help the process. There are also some criteria to be taken into consideration. First of all, the need to integrate at all should be determined. If it is possible to develop a single application that does not need any collaboration with other applications, trying to implement an integration solution would only needlessly complicate the architecture and possibly slow down the system. Realistically, though, enterprise applications almost always need some kind of integration. [1, p. 1]

To determine which kind of solution to implement, some of the most common criteria are listed by Hohpe and Woolf [1]:

- Application coupling – The less applications depend on each other, the better.
- Flexibility – How much software and hardware is required by default? How easy it is to do changes for the integration solution?
- Data format – Applications must agree which type of data to exchange.
- Performance – How long does it take for the shared data to reach its destination.
- Sharing method – Do the applications share data or functionality? Sharing functionality can give better abstraction between applications.

- Synchronization – Is the sharing synchronous or asynchronous? Asynchronous connection is usually better, for not having to wait for answer, but it is also more complex.

- Reliability – Remote applications may not be always available or even running at all. Integration solutions should take this into account.

There are four main approaches to developing an integration solution: file transfer, shared database, remote procedure invocation, and messaging. Each approach has its own strengths and weaknesses, and each of them apply against the criteria differently. Also, each solution builds on the previous one, adding more features and possibilities of use, but also making them more complex.

File Transfer (Figure 2.3) is the most simplified integration approach. In it, the applications write files to a predefined location, where other applications can read them. The files must be in a specific format and the applications must know when to read, write and delete the files. That way minimal amount of software and hardware is needed, but also a lot of manual work is needed defining all the names and locations. As for the file format, XML is commonly used due to its standardization. The files are usually produced and consumed at fixed periods (daily, weekly, monthly), which means that the systems can get momentarily out of sync. Getting out of sync is indeed the biggest problem of file transfer, since it requires developers to constantly fix any inconsistencies. Application coupling is one of this approaches strong points, since there is only minimal knowledge of other applications. [1, p. 43-46]

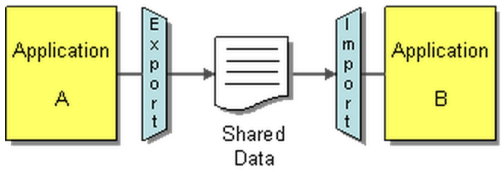

Figure 2.3: File transfer moves files from application A to application B. [1]

Shared database (Figure 2.4) improves on the one major drawback that file transfer has, that is the data inconsistency due to synchronization problems. It uses a single database schema in a single location, which multiple applications can then use. Thus no data needs

to be transferred between applications, and the data is available instantaneously. Also, since the data is in one single location, it is always consistent. File format will not be a problem, because almost all platforms can work with SQL. One of the problems of shared database is designing an unified database that suits the needs of multiple applications. This usually makes the database complex and hard to use. Another limitation is the adaptation of the database schema. A lot of applications will not work with a schema other than their own. Finally, using a single location for the data may cause performance issues when multiple applications are trying to access the same data. [1, p. 47-49]

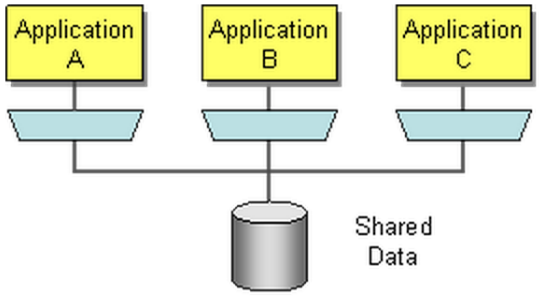

Figure 2.4: Shared database uses one database for multiple applications. [1]

Remote procedure invocation (or Remote procedure call, RPC) is needed, when exchanging data is not enough. Changing some piece of information may require a chain of actions that need to be done. However, giving straight access to these actions breaks the idea of loose coupling. Some kind of interface is needed in order to invoke needed functions. RPC does exactly that, and can be seen in Figure 2.5. Data is encapsulated through an interface, and each application can process the data as they want. The communication is synchronous. Remote procedure invocation is implemented by many technologies, such as Java Remote Method Invocation (RMI), COBRA, COM and .NET Remoting. Also, RPC can be used with Web Services using technologies such as SOAP and XML. One of the advantages of RPC is its flexibility. Applications can implement multiple interfaces, making it possible to allow multiple ways of accessing the data. On the other hand, RPC has a disadvantage of being too tightly coupled despite the encapsulation. It groups actions into sequences, which are always done in a certain predetermined order. Thus it is hard to make any changes to the individual applications. RPC might also have problems with performance and reliability. [1, p. 50-52]

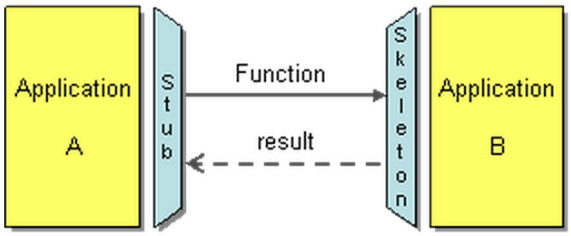

Figure 2.5: Remote Procedure Invocation shares functionality instead of data. [1]

Messaging is required to achieve more loosely coupled and asynchronous solution. Messaging (Figure 2.6) focuses on asynchronously sending small packages through common channels to publish and receive messages between applications. The messages can be transformed on the way, without applications ever knowing of it. Thus the same message can be sent to multiple destinations, and in each destination the message is transformed into a fitting format. The applications send small packages frequently, and the communication is fast and reliable. Still, there are some problems with messaging too. The messages might produce some lag, and testing and debugging are harder. Also, a lot of factors need to be considered, such as how the data is transferred, where it is sent and in what format. Most Enterprise Application Integration (EAI) implementations use this approach as it provides the best results for the given criteria. [1, p. 53-56]

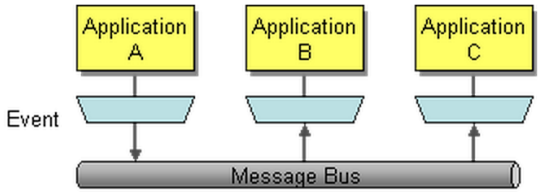

Figure 2.6: Messaging relies on sending data packets through a message bus. [1]

Developing an integration solution is never an easy task. EAIIC European Chairman Steve Craggs lists 7 most common challenges that developers meet: constant change, complexity, lack of universal standards, ability to treat EAI as more than just a simple tool, building an interface that everyone agrees on, keeping all the required details in the solution, and managing corporate policies [10]. Almost all of these are management issues, rather than technical problems. Constant change of the integration solutions is caused by the needs of businesses, which change constantly. Therefore, the solutions must be flexible enough. Complexity comes from possibly handling a large system consisting of asynchronous processing and multiple combinations for data. Adding to that, the lack

of universal standards may confuse developers even further. There are a lot of different standards, which sometimes collide with each other. Ability to treat EAI as more than a simple tool means that the integration solution is not just a tool to transfer data, but a complex system, which has multiple functions, such as load balancing and security issues. When building an integration solution, it comes critically important to build a system which provides sufficient interface for all the participants. Otherwise there might be some excessive mapping between the data on different systems. Also, some details which seemed unimportant earlier in the development stage, may become important later. Lastly, integration solutions usually deal with multiple business units and IT departments. This can cause some maintenance problems, due to multiple departments having to work together.

2.4. Enterprise Integration Patterns

The idea of using patterns as a guide and a language to design software solutions came first from Christopher Alexander in 1987. It revolves around the idea that after a software designer has come across a lot of similar type of problems, over time he or she develops some habits, or "patterns", on how to react to them. These patterns are designed to solve some specific problems in a specific situation, or to describe some good design practices. This same approach can be used to enterprise integration. [11]

Enterprise Integration Patterns (EIP) are based around messaging, and currently consist of 65 distinct design patterns. Their goal is to help developers apply enterprise integration as efficiently as possible. They are not technology or platform dependant, but more of a higher guideline to integration. EIP introduce patterns, such as Message Channel and Message Broker. In order to understand them, basic concepts of messaging systems need to be understood first. [2]

Messaging systems consists of messages, channels, pipes and filters, routing, transformation, and endpoints. The first problem to solve is how to connect multiple applications using messaging. Applications cannot randomly send data to a messaging system. They must know where and what type of data to send and receive. For this purpose, Message Channels (Figure 2.7) are utilized. Message Channel is a virtual pipe, which transfers data from one end to the other. Applications do not necessarily need to know where the

data is going, but only that the data is ensured to reach its destination and that the application on the other end is interested in the data. Since channels are logical addresses, the actual implementation of them is hidden from the application and depends on the messaging system. The channels that are needed in the application are usually configured while installing the messaging system. One thing to remember when creating channels, is that each channel requires a certain amount of memory and possibly disk space, so they should be added only when needed. As for the types of channels, there are two main approaches: Point-to-Point and Publish-Subscribe channels. Point-to-Point channel transfers messages from one end to another, while Publish-Subscribe channel can send messages from one point to multiple locations. [1, p. 60-63]

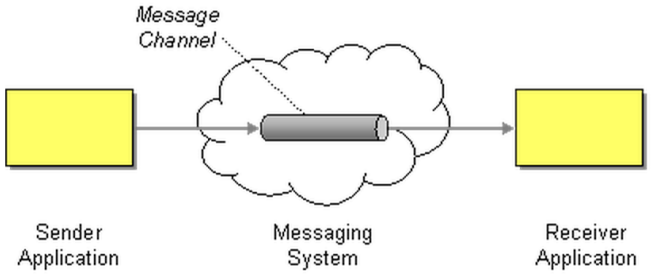

Figure 2.7: Message Channels. [1]

To send data through channels, some units of data must be used. In messaging system this is done through messages. Messages (Figure 2.8) consists of two parts: a header and a body. Header has all the metadata about the message, like its origin and destination, while the body has the actual data of the message. In order to send messages, the data must be first transformed into a byte stream. This is called marshalling. After that the message is sent to the receiver, which must unmarshal the message back to its original form. Messages can be used for multiple different purposes. Different types of messages include command, document and event messages for invoking procedures, sending data and informing of a change. [1, p. 66-67]

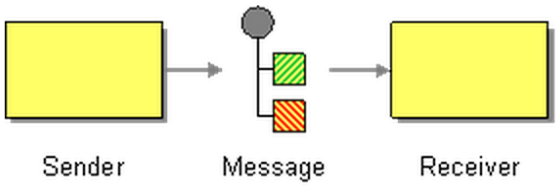

Figure 2.8: Messages. [1]

Sometimes simply sending a message trough a channel is not enough. Some kind of processing may be required, for example an authentication. This can, of course, be achieved by implementing an authentication check on the application side for all the required applications. However, doing this would be redundant and inflexible and would make the integration solution more tightly coupled. A better solution would be to add the processing steps to the messaging system as filters, connected by channels (pipes). This can be seen in Figure 2.9. Each filter takes a message, processes it in some way and sends it through the next pipe to its destination. Due to messaging being asynchronous, multiple messages can be processed at the same time. For example, if the messaging system has two filters, decryption and authentication, both of these filters can be processing a different message at the same time. This, of course, means that the process is hindered by the slowest filter in the chain, since the other filters have to wait for it to finish. This can be improved with parallel processing, adding parallel filters for the slowest one. Problems can occur with this architecture, if there are a lot of pipes and filters. Then the pipes can consume a lot of memory and the filters may cause performance problems, because the messages have to be translated between application's data format and the messaging system's format for each filter. [1, p. 70-74]

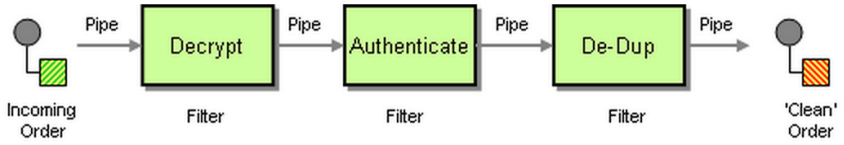

Figure 2.9: Pipes and Filters. [1]

The pipes and filters architecture connects each filter with a single pipe, making the message go through the same steps every time. However, often different type of messages require different kind of processing. A simple solution would be to implement a message channel for all the different messages, but this could create unnecessarily large amount of channels. Instead, a new special filter can be added for the pipes and filters architecture. This filter, called messaging router (Figure 2.10), takes a message and sends it to a destination, which is determined by a set of conditions. Thus the router can be connected to multiple output channels, unlike normal filters. Having a single filter for all the routing makes maintenance of the routes easier. If changes are made, only one component needs to be updated. However, if the destinations change frequently, maintenance may become

increasingly frustrating. A message router can also become a performance bottleneck, sine all the messages must go through it. The type of the router can vary regarding how it processes the message. The most common case is to perform routing by the message content, but it can also look for the message context, for example. [1, p. 78-82]

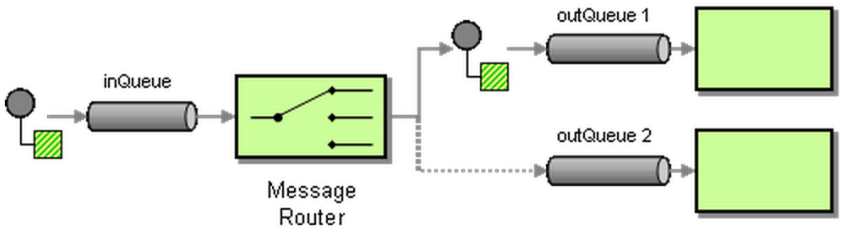

Figure 2.10: Message Routing. [1]

Often different applications will not be using the same information in the exact same format. For that, some mechanism is required to transform the messages from the sender to a format that the receiving application can use. The translation can happen on many levels. On a simple case, data may require only some field name changes. However, sometimes the data may be required to translate from one format to a completely different, for example from XML to CSV. If transformation is needed on multiple levels, they can be chained with multiple message translators. For transforming XML files, W3C has defined a standard language XSL. Message translator can be seen in Figure 2.11. [1, p. 85-90]

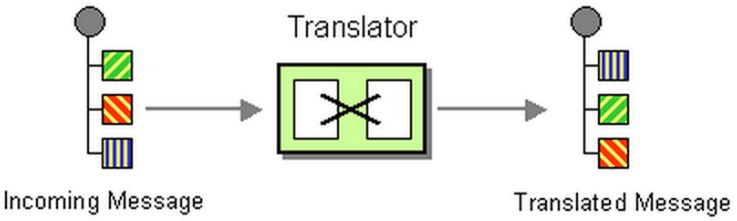

Figure 2.11: Message Translation. [1]

Finally, a question remains on how the applications can communicate with the messaging system's API. A message endpoint (Figure 2.12) can be used for this purpose. The sender application gives one endpoint some data, which the endpoint can send to the messaging system. The message is then delivered to the receiving application's message endpoint, from which the receiving application can ask for the data. That also means, that the endpoints hide the messaging system from the applications. [1, p. 95-97]

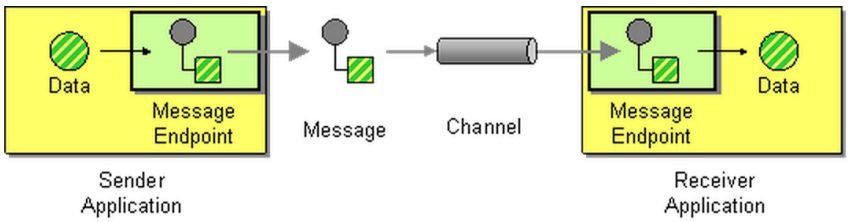

Figure 2.12: Message Endpoint. [1]

3. INTEGRATION FRAMEWORKS

To implement an integration solution, three possibilities exist. Firstly, one can build one's own custom solution, thus building everything by oneself. This is probably the fastest solution for smaller cases, where only a very light integration is needed. However, this will be very time consuming, if the use case is a bit more complex. Also, maintenance can be tough for someone else than the original developer. The second solution is to use integration frameworks. They were created to help build integration solutions easier and in a standardized way. Integration frameworks provide a software platform and a set of APIs to develop integration solutions. There are many possibilities to choose from, and all of them have their own strengths and weaknesses. The third option is to use an enterprise service bus (ESB). It adds even more functionality to the integration solution. ESBs have the same functionality as integration frameworks, but they can also take care of business process management or monitoring for example. Each of the three options add complexity to the previous one, but they also add functionality and give more possibilities. This chapter introduces some of the most common integration frameworks and ESBs. [12]

3.1. Apache Camel

Apache Camel is an open source integration framework based on the EIPs. It has great connectivity to other systems and its own Domain Specific Language (DSL) to tie everything together. High level view of the architecture of Camel is shown in Figure 3.1. Some popular projects, that are often used with Camel include ServiceMix (ESB), ActiveMQ (message broker), CXF (web services suite), Karaf (OSGi based runtime) and MINA (networking framework).

Components are the connection points of Camel to other systems. They are used to create Endpoints in Camel. At its core, Camel includes only 13 components. However, it has over 80 components outside the core for more specific cases. Components have

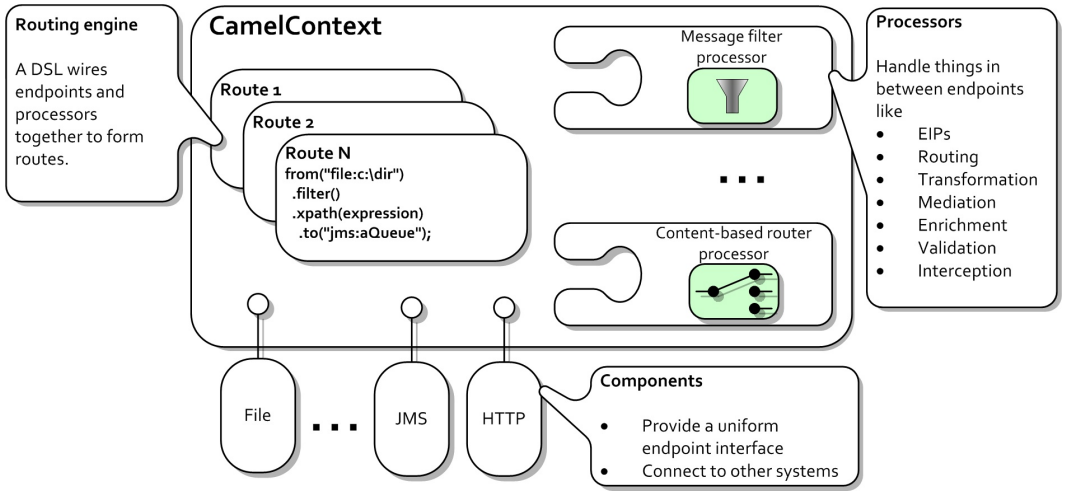

Figure 3.1: Architecture of Camel. [13]

also a message endpoint to make them work with the other parts of Camel. Different types of components can be used together for example to take a message from one type of component and send it to another. Processors handle the processing of the messages. After receiving a message from one endpoint, they do any necessary filtering, routing, etc. and send them to the other endpoint. The processors are taken straight from the EIPs, currently supporting most of the patterns. To connect the endpoints to processors, DSLs are needed. Some possible languages are XML, Java, Scala and Groovy. Examples can be seen in Programs 3.1 and 3.2, where we create JMS messages from file contents in the given path and send the messages to the JMS queue called "queue". [13]

Program 3.1: Camel DSL for XML

```
<route>
  <from uri="file:/incoming"/>
  <to uri="jms:queue"/>
</route>
```

Program 3.2: Camel DSL for Java

```
from("file:/incoming").to("jms:queue");
```

3.2. Spring Integration

Spring Integration (SI) is an open source integration framework, and like Camel, it too is based on EIPs and implements most of the patterns. SI builds on the existing support of enterprise integration for the Spring framework and thus is easily implemented to projects

already using Spring. The Spring framework is an open source application framework for Java platform. [14]

Spring Integration has support for the basic transporting technologies, such as File, FTP, JMS, TCP, HTTP and Web Services. They are called adapters in SI, and are equivalent to components in the Camel architecture. The architecture of SI is very similar to the Camel architecture, shown in Figure 3.1. One difference is that the SI uses only XML to implement the integration. An example can be seen in Program 3.3, which does the same thing the earlier Camel example did. [15]

Program 3.3: Spring Integration example

```
<file:inbound-channel-adapter
  id="incoming"
  directory="file:incoming"/>

<jms:outbound-channel-adapter
  id="out"
  destination="queue"/>
```

3.3. Enterprise Service Buses

Enterprise Service Bus (ESB) can be defined as a software architectural model, that different systems in an IT environment can use to communicate with each other. Compared to integration frameworks, ESBs offer more features in a single package. Thus integration frameworks are usually more lightweight, but more limited. All the extra components can be added separately to build a custom ESB, but using a predefined ESB can be easier to set up and maintain. ESBs can offer message processing related features, such as transforming, routing and enhancement (filling missing data). On top of this, it also adds abstraction between locations, supports multiple transport protocols, enables authentication, authorization and encryption, and helps monitor and manage the integration system. [16]

Before ESBs emerged as an integration solution, EAI solutions used hub-and-spoke architecture, more commonly referred as the message broker, which had EAI broker as the center of the solution, connecting everything. This is also listed as one of the enterprise integration patterns. The centralized hub is capable of processing the messages (transformation, validation, routing) and send them forward to their destinations. Thus it

leaves the connected systems almost untouched. Disadvantage using hub-and-spoke is, however, that it puts some heavy load on the broker and set it as a single point of failure. If it goes down, the whole messaging system goes down with it. Advantage compared to a point-to-point architecture (or the so-called integration spaghetti) is that hub-and-spoke improves management considerably and requires less nodes. Both architectures can be seen in Figure 3.2. [17]

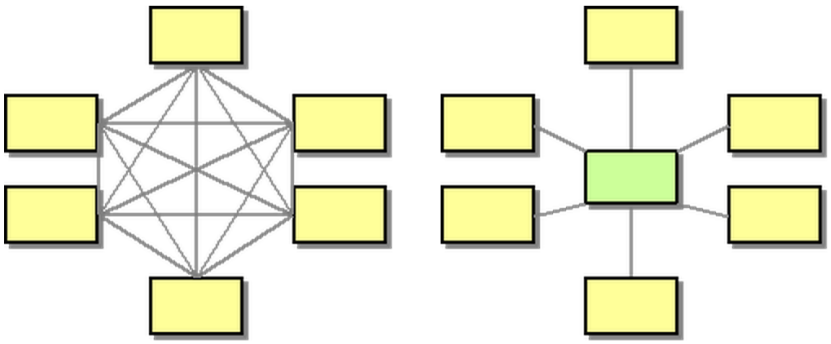

Figure 3.2: Point-to-point integration (left) versus the hub-and-spoke architecture (right). [17]

To evolve from this, ESBs were developed. ESBs have a logical bus between each system connected, decoupling the systems from each other and adding a level of abstraction. Different systems need only to know about the ESB, and not necessarily about any other systems. Architecturally ESB looks very similar to the hub-and-spoke architecture. Different systems are connected to the ESB, which routes messages to their correct destinations. Message processing, such as transformation, is done in the connector between the ESB and the system. ESB architecture can be seen in Figure 3.3. The key difference between ESB and hub-and-spoke architecture is, however, that the ESB architecture is based on SOA, so that the components in it are distributed across the bus and hosted in separately deployable containers. This allows the usage of only the needed components for each system, reducing the overall load compared to hub-and-spoke and improves scalability. This underlines the other key difference, which is that hub-and-spoke is centralized, whereas ESB is distributed. Also, ESBs are based on open standards. [18]

Mule ESB is an ESB, which, like Apache Camel and Spring Integration, is based on the Enterprise Integration Patterns. Unlike the two though, Mule is a full ESB instead of being just a framework. It can still be used as a lightweight integration framework by

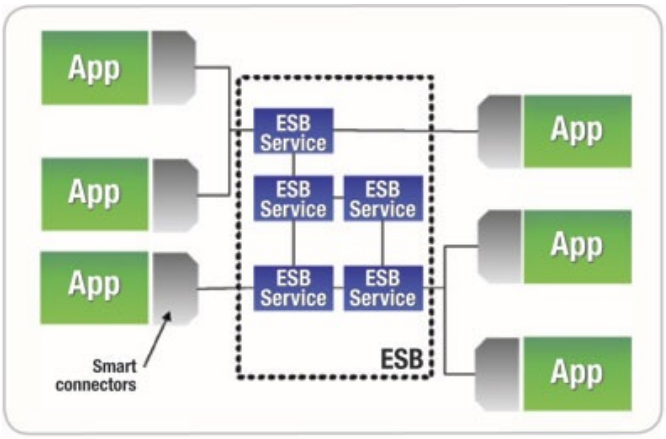

Figure 3.3: ESB architecture. [18]

just leaving out all the additional features it offers, and use only the EIP based integration module. Mule shares a lot of similarities with Camel and SI. It is open source and designed to be lightweight and scalable. It should be noted though, that it does not have a truly open community, as MuleSoft makes the final decision on what to implement. As for the DSL, Mule offers only XML, same as SI. Example can be seen in Program 3.4. [15]

Program 3.4: Mule example

```
<flow name="muleFlow">
  <file:inbound-endpoint path="incoming"/>
  <jms:outbound-endpoint queue="queue"/>
</flow>
```

Flows in Mule represent the same thing that routes do in Camel (Figure 3.1). They can contain inbound and outbound elements to configure connectivity. Components in Mule are not type restricted. They can be implemented with any number of technologies, including Java and Spring. As for connectivity, Mule offers more than 20 transport protocols and can be integrated with projects such as Spring and ActiveMQ. The architecture of Mule was never based on Java Business Integration (JBI), unlike many others, but instead built on a flexible and lightweight model. Also, it does not support OSGi, claiming it is great for middleware vendors, but terrible for end users [19]. Mule has also emphasized component reuse and does not put any restrictions to message format. [16]

ServiceMix was introduced in 2005, and it was first built on JBI specification. Later on, though, when it was determined that JBI will not receive any more updates, ServiceMix

moved to OSGi based solution [20]. OSGi is a Java based framework that helps to manage (install, update, start, stop) different modules in the system. The modules are called bundles. At its core, current ServiceMix (ServiceMix 5) is running Karaf, an OSGi runtime that handles management and deployment.

ServiceMix uses Camel as basis for implementing an EIP based ESB. For default messaging (JMS) provider, it uses ActiveMQ, but it can interact with other providers too with a matching connection factory. ActiveMQ provides reliability to the messaging and allows a distributed environment. Other technologies that ServiceMix can be integrated with, include Apache CXF, Apache ODE, Apache Geronimo and JBoss. ServiceMix is largely used around the world and has an active community. Some of the core developers of ServiceMix and ActiveMQ moved to later develop Fuse ESB, a ServiceMix based ESB with minor differences. [16]

OpenESB is an open source ESB started by Sun Microsystems. Since Oracle and Sun merged, however, a separate OpenESB community was born to maintain an develop it. OpenESB is based on the JBI specification and has not moved out of it like many other ESBs. It provides lightweight and scalable JBI implementation. JBI's purpose is to define a standard for an integration platform. It tries to prevent vendor lock-ins by having components from open source projects. OpenESB includes many subprojects, such as JBI runtime environment, JMS binding, and BPEL (Business Process Execution Language) service engine. [21]

OpenESB differs from other ESBs by having a strong focus to integration with Glassfish. It provides tools and wizards for building JBI components and dependencies between them. Also, it provides editor for BPEL and WSDL among others. For message provider, it uses OpenMQ as default. Despite the lack of updates to JBI, OpenESB has still been developed further. [16]

Talend is an open source ESB, which, like ServiceMix, uses many Apache products as its basis. These include Camel, CXF, Karaf and Zookeeper. In addition to the connectors that Camel provides, Talend also includes some other adapters, such as Alfresco, Jasper, SAP and Salesforce. It is an extensive ESB, having many features and management tools in addition to the core functionality. [22]

The free version of Talend provides all the basic functionality, but there is also a propriety version, which offers additional features and support. Tooling support is available in both versions. The tooling is built on Eclipse. Talend also offers its own visual editor for designing the integration solutions. It is possible to create the solution with the designer without coding anything. Talend ESB is part of a larger Talend suite, founded in 2005, which is focused on data integration, data management, EAI and big data.

Petals is developed by OW2 consortium, which is non-profit, independent open source software community. Petals itself is an open source ESB which emphasizes on SOA. It has been developed since 2006, and is usually suitable for large scale SOA solutions. What makes it different from others, is its distributed aspect. Petals can be physically constructed from many servers, but will seem like a single bus. This works well with SOA principles and gives some flexibility for the architecture. It supports many of the common communication protocols.

Like OpenESB, Petals is also based on JBI specification, which is a major drawback for it. Development is done through an Eclipse plugin. There is also a JBI component framework on top of that. Aside from that, it gives high availability, failover capabilities and good scalability. [23]

WSO2 ESB is a little less known ESB on the market, but it is gaining popularity gradually. It is developed by WSO2, an open source application development software company, founded by Dr. Sanjiva Weerawana in 2005. The ESB is marketed as being lightweight and having high performance. It often has good results in performance tests [24]. A popular example of a product using WSO2 ESB is eBay, having to process over 1 billion transactions daily. The ESB is open source like most of the others mentioned. It is built on top of WSO2 Carbon platform, which is an OSGi based framework. [25]

As for its features, WSO2 provides good mix of transport and connection protocols. On top of that, it also offers tools for a full integration suite, such as Business Process Server and Business Activity Monitor. The development can be done with an Eclipse-based tool. It also includes a graphical development tool, which, according to Wähner, leaves some space for improvement, since it is not very intuitive. [22]

UltraESB is a lightweight and fast performing ESB released by AndroidLogic in 2010 [26]. Like WSO2 ESB, it has not gained that much popularity, but has been performing well in any performance tests. It tries to ease the development and debugging by supporting the popular IDEs (IDEA, Eclipse, Netbeans). To improve the performance, it uses things like file caching on RAM disks.

The main features that distinct the product from others are: fast performance, easy extensibility with custom components, support for many transport protocols and a lightweight build. The main problem with the ESB seems to be that it is licensed under Affero General Public License (AGPL) [26], which does not allow production use for any closed products.

4. INTEGRATION CRITERIA FOR VALTIMO

To choose the best integration framework for a specific situation, a certain set of criteria need to be explored. These may vary widely, and they weight differently based on how important they are to the solution. In this chapter, these criteria will be selected for Valtimo. To measure how the products fulfil the given criteria, an evaluation model will be created based on them. This helps to pick the best integration framework for our solution. There is also the question of whether to use an ESB or not. Using ESB gives more features available in a single package, but it may also slow down the solution by adding some unnecessary layers.

4.1. Project Valtimo

Occupational safety and health administration of Finland does roughly 25 000 labour inspections and gets more than 70 000 contacts yearly. To help manage the processes involved in the occupational safety control, project Valtimo was created, and the ministry of social affairs and health assigned Gofore to do a large part of it. Aside with the produced management tools, a new nationwide operating model is also created. The project began in 2010, and the earliest versions of the produced system were released and put into use in 2011, as the first of the six subprojects was finished. The final part was started in august, 2013, and the whole Valtimo project should be ready in 2014. As a result, controlling occupational safety should become much easier and more efficient, as all the data is stored in one location using the produced system. [27]

As for the architectural structure of the system, the project was designed with enterprise architecture and Service-Oriented Architecture in mind. The different layers and their connections to each other can be seen in Figure 4.1. The system is divided into three main layers: presentation, business logic, and database. The presentation layer includes the user interface and the logic related to it. User identification is also done in this layer,

and the content is shown based on which role the user has. Different roles have also different levels of authorization to operations in the system. Role authorization spans through every level of the architecture. The business logic (or service) layer deals with fetching data from database and executing all the actions the user does. To connect these two layers, there is an ESB between them. The ESB takes XML messages, and routes them from one end to another based on some routing rules. The ESB has also a service interface (based on WS-I Basic Profile), which transfers messages to the service layer. It is also possible to add other integration solutions to the ESB. Lastly, there is the database, which is connected to the service layer, and contains all the persistent information. Presentation layer does not have access to the database. [28]

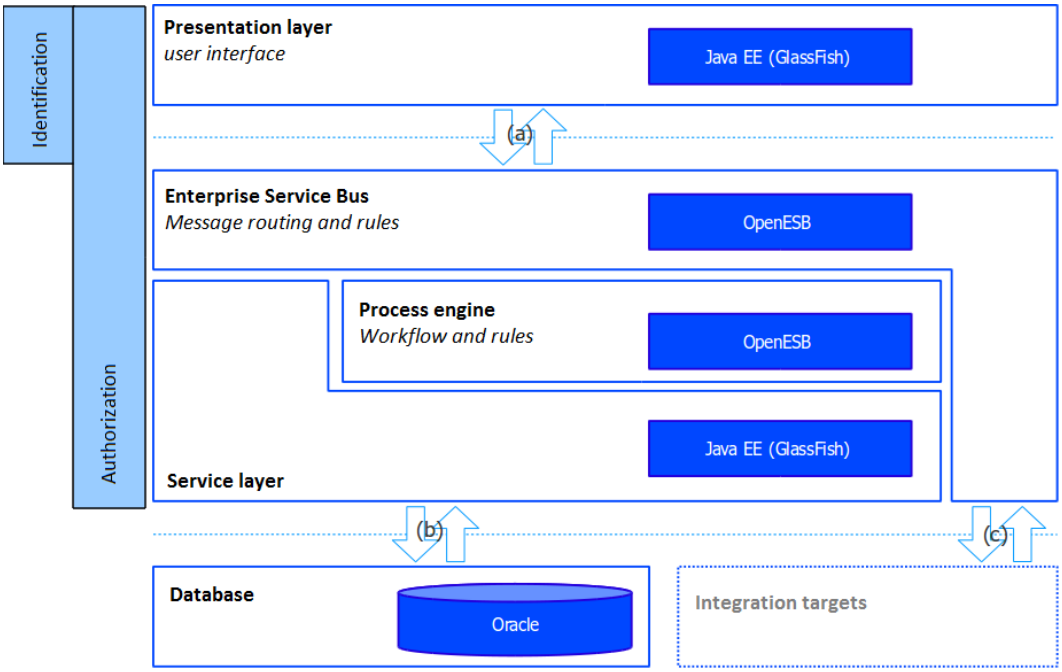

Figure 4.1: The basic architecture of Valtimo.

The main technologies used in the system are also shown in Figure 4.1. The programming language used is Java and the platform Java EE, more specifically Java EE 6 in the current versions. It is used with Glassfish, which is an open source application server project for Java EE. Enterprise JavaBeans (EJB) is used as a part of it for the service layer. For data storage, Oracle Database 11g is used with Java Persistence API (JPA). For the user interface, Valtimo uses JavaServer Faces (JSF) with PrimeFaces.

The integration in the system is implemented using OpenESB, and the data flow can be seen in Figure 4.2. The presentation and the service layer are both connected to the ESB

with OpenMQ, which only forwards the messages from one queue to another. Thus the system has only four fixed queues, one for incoming and one for outgoing for both ends. On top of this, the ESB has also a WSDL interface for external client software. It routes the messages to the service layer and gives the response back to the requesting client. All the messages used in the system are XML-based, and the technology for sending messages is JMS (Java Message Service). [29]

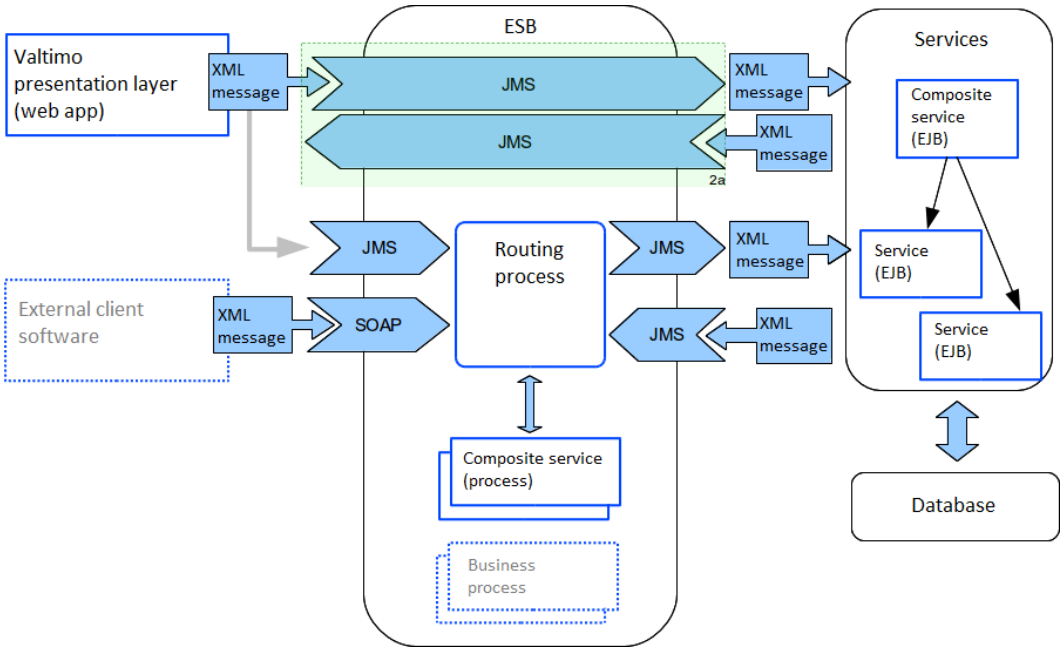

Figure 4.2: The integration and data flow in Valtimo.

The main reason why this integration solution needs upgrading is that OpenESB uses JBI (Java Business Integration) specification. It seems that JBI does not offer sufficient specifications, and further development of it has been withdrawn as of 17 Dec 2010, when the JBI 2.0 was cancelled [30]. Several ESB solutions, like ServiceMix, Fuse and Mule, have since been moving out from JBI to some other options. Also, there are other ESBs that are updated more frequently, and can thus be safer picks for the future than OpenESB. There might also be a possibility to rethink some of the integration related decisions in the system. [20]

4.2. When to use Enterprise Service Bus

The first question that arises, is whether to use an ESB, or just an integration framework with a message broker or a smart endpoint. An integration framework is a lighter solution.

It can integrate solutions with different transfer protocols and technologies. To connect applications and to create the logic, it uses standardized libraries and functions (such as Message Producer and Message Consumer), and thus works with any environment. There is not much tooling support for it, and the developers need to create a lot of the code by themselves. An ESB can also do the integration between applications, but it adds to a framework by giving a more complete package with less self written code and better tooling and management support. There can also be commercial support, whereas the introduced integration frameworks have none. For even larger systems, there are integration suites, which offer even more features, such as Business Process Management (BPM) and Business Activity Monitoring. [22]

To make the decision easier, Ross Mason, the founder of Mule ESB, has made a checklist of things to take into consideration [31]:

- Are you integrating 3 or more applications/services? If not, point-to-point integration is easier.
- Will you need to plug in more applications in the future? There needs to be a real reason why there might be additional applications later. Doing it for 'just in case' is not a good plan.
- Do you need to use more than one type of communication protocol? Using only one, the benefits of cross protocol messaging and transformation of ESBs will not be utilized.
- Do you need message routing capabilities such as forking and aggregating message flows, or content-based routing?
- Do you need to publish services for consumption by other applications?
- Do you need scalability of an ESB? It is easy to overestimate the needs to scale an application.
- Do you really understand what you want to do with your architecture? ESBs have a lot of features that need to be understood in order to use them in full effect.

This list should give some idea of what kind of things should affect the decision. In the case of Valtimo, we are integrating only 2 applications, but there is also a requirement of a web services interface, making the system a bit more complicated. As for future needs, there is a confirmed requirement for an additional integration point. Used communication protocols are JMS and Web Services. The need for routing capabilities are not yet clear, but they might be needed. Publishing services will not be required. The application will be used all over Finland, having several calls every day.

Taking into consideration all the given criteria and the needs of the architecture, using an ESB seems to be a better choice in this case. If only integration framework is needed in a project, Camel seems to be usually the strongest choice. It is also the most popular one of them all. Spring Integration comes naturally into use, if the application is already built on Spring.

4.3. Criteria for ESBs

There are several criteria that affect the decision for the most suitable ESB for a project. Some of them are technical, while some are more related to project management. The following list will include some of the most important criteria. They are ordered by their relevance to Valtimo. These criteria are a combination of things that have been mentioned the most in multiple different sources. Taking advantage of the steps used in the QSOS (Qualification and Selection of Open Source software) method developed by Atos Origin and discussed in Rautonen's thesis [32], the criteria will be further filtered to take into account only the ones that are relevant to the Valtimo project. After this, the remaining criteria can be weighed, and the products will be given points for how well they do against each remaining criteria. The product with the most points will be used in the new integration solution for Valtimo.

4.3.1. Core functionality

The first criterion is about the core features that every integration solution should be able to handle. These include some basic functionality for messaging. First of all, multiple transport protocols should be allowed to be integrated together so that a service using one transport protocol can be connected to a service using another protocol. Solutions should

also be able to handle the common EIP based features, such as message transformation and routing.

There are also some issues concerning security, management, and transparency. Messages should be able to be authenticated and authorized in order to prevent malicious activity. Encryption may also be needed. Since the integration solutions are usually rather large, some kind of management and monitoring is required. This can help notice problems any in the system, for example some parts not responding to any messages or just performing slow. Lastly, there is the issue of location transparency. In order to enhance SOA like behaviour, different parts of the system do not need to know the exact location of each other. Rather, they only need to know about the integration solution, which routes the messages to their destinations. [16, p. 13-20]

Since ServiceMix, Fuse and Talend are all based on Camel, which in turn is based on EIPs, they handle most of the messaging related requirements well. Mule is also based on EIPs, so it is on the same level as the ESBs using Camel. OpenESB is not based on EIPs and it does not have all the patterns implemented. They can, however, be mostly implemented with BPEL [33]. Petals provides an implementation for some of the patterns, but not all [34]. WSO2 includes a documentation on how to implement each of the patterns with their product [35]. The last of the group, UltraESB, implements most of the patterns too [36]. For the security, management and transparency criteria, all the ESBs are performing almost equally well.

4.3.2. Open source

When choosing a framework or ESB, a question rises whether it should be open source or not. There are several advantages and disadvantages with both approaches. When choosing a proprietary solution, there is constant enterprise support available and a rich set of functionality. There are usually also some powerful tools for management and monitoring. However, the community might be lacking, and the solutions may be hard to use and lack any extensibility and flexibility (unless there is a possibility to pay for new features). Also, the cost is high and licensing may cause issues. [22]

Open source software systems, however, are free to be used and are based on less restricting licenses. They have usually a stronger community working to improve the

software and help people in need. Open source projects are also usually more flexible and may allow people to add their own improvements to them. Downside is that there may not be as good support for them, and they might not have as many special features as proprietary solutions. Also, the monitoring and management tools may not be as good. Still, based on these criteria, the open source solutions seem to be much better for Valtimo than proprietary solutions on most parts, and they are also much more popular. [22]

Almost all of the ESBs in the comparison group are open source. Fuse has been recently bought by RedHat, and while it is considered open source, it requires a commercial license for production use [37]. Mule markets itself as an open source ESB, but it has had some criticism of not truly being one, since MuleSoft owns the product and can make the final decision on what to implement in the ESB.

4.3.3. Licensing

Licensing is a factor, which may restrict using a product in certain context. Ultimately it may prevent developers from choosing a tool altogether. Usually, the open source ESBs have a license, which allows developers to use them in production without many restrictions. Most of the times the open source licenses are free of charge. Propriety ESBs have a bit stricter licenses and are rarely free. For Valtimo, it is important to choose a tool with a licensing, that does not restrict the development in any way.

ServiceMix and WSO2 use ASL (Apache License) 2.0, which gives exclusive rights to use the product and does not pose any larger restrictions [38]. Mule uses CPAL (Common Public Attribution License) for the community edition, which also allows development without much restrictions [39]. For the enterprise edition, Mule has a commercial license. Fuse uses ASL 2.0 for development purposes, but for any production use, a commercial license is needed [37]. OpenESB uses CDDL (Common Development and Distribution License) 1.0. It allows production use without any cost or any larger restrictions [40]. Talend uses ASL for the community version, and a subscription licence for the enterprise edition. Petals is licensed under LGPL (Lesser General Public License), which is a free software license [41]. UltraESB is licensed under AGPL (Affero General Public License), which restricts any proprietary use with a closed product [42]. This effectively renders the product useless for Valtimo.

4.3.4. Popularity and future

Popularity of the ESB is an important factor when looking at the life expectancy of the product and activity of the community. Popular ESBs are likely to live longer and have a strong community developing it further. Life expectancy is important considering the future maintenance of the product. Having to change the ESB later may require a lot of work and bring unnecessary costs. Activity of the community is more important in the development stage of the application. Community can help developers in any occurring problems and may help fixing any bugs or lack of features in the ESB itself. Since this criterion is one of the reasons that the current integration solution in Valtimo is being changed, it is important to take this criterion into careful consideration. [16]

Currently, according to Mulesoft, Mule ESB is the world's most widely used ESB [43]. This is probably followed by ServiceMix, which has a strong and active community. Fuse is also popular, but the recent acquisition by RedHat poses a little risk for the future. OpenESB is slowly fading away and is not keeping up with the rest of the ESBs [44]. Using JBI further hinders its future prospects. Same goes for Petals. Talend is not quite as popular as ServiceMix or Mule, but it is still having a strong community. WSO2 is gradually gaining more popularity, but is not yet there with the most popular ones. UltraESB is rather less known ESB.

4.3.5. Enterprise readiness and market acceptance

Enterprise readiness here means how mature the ESB is as a product and how well it is accepted by the community. Products which have been around longer have had time to process and improve their ESB solution and develop it into a more complete package. It usually also means that a larger audience have been introduced to the product and have used it. Open source ESBs usually take a lead against proprietary ones in market acceptance, since they are more accessible. [16]

ServiceMix and Mule seem the two most accepted products on the market today. ServiceMix has gained a large user base by using well known components, such as Camel and ActiveMQ, as part of its core. Mule has been around a long time and have had time to improve the ESB and secure its place as one of the more used ESB platforms. Fuse is somewhat used, but not quite as mature as the previously mentioned ESBs. OpenESB's

problem is JBI, which has lowered its market acceptance by a large margin. The same goes for Petals. Talend has some acceptance in the market, but it is not quite as matured as some of the others. WSO2 is slowly gaining market acceptance and is becoming one of the more well known ESBs. UltraESB has not yet gotten a wide market acceptance, mostly probably because of its license.

4.3.6. Expandability and flexibility

One important factor when choosing an ESB is how easy it is to write any custom logic to expand the current features of the ESB. Often projects have some unique requirements, which the ESB cannot handle on its own. Thus it should be as effortless as possible for a developer to write some own code to handle those specific cases for a project. Valtimo may require some expanding in the future, so this criterion needs to be taken into consideration. [16]

Most of the ESBs allow user to write some custom logic in them. ServiceMix and Mule enable developers to use POJOs to write any extensions, such as custom components, thus making the customization easy and accessible [16]. Talend offers some good extension possibilities as well [22]. WSO2 ESB has possibility to create custom mediators to write any custom logic [45]. UltraESB has also custom mediators for any needed logic [46]. They can be made with Java or some scripting languages. The other ESBs have not focused on those aspects and do not have much information on how to create custom logic in them.

4.3.7. Connectivity

Connectivity as a criterion determines how well components can be integrated to other components with different technologies. An ESB should provide necessary endpoints and transporters in order to make the required integration routes without having to make any custom transporters, which is usually rather tricky and requires some expertise. Because of the multiple ways of communication in Valtimo, this criterion is the final one that will be included in the evaluation process. [16]

ServiceMix, Fuse and Talend are based on Camel, which by default has a wide variety of connection options and endpoints. Talend has also some additional adapters [47].

Mule, WSO2 ESB and UltraESB provide rather good connectivity support too. Petals supports most common connectors and has tools to create custom ones. The rest of the ESBs do not stand out on this category.

4.3.8. Commercial support

If the help of the community is not enough, sometimes ESBs offer some commercial support in addition. It can be more efficient way to solve any problems, and a solution is usually guaranteed to be found. It can also provide help in reviewing the overall architecture. The price of the services may vary widely.

ServiceMix offers commercial support through a few consultant companies, Ameliant and Savoir Technologies Inc [48]. For the Fuse ESB, commercial support is offered through Fusesource's subscription option, which will offer a wide variety of tools and documentation [49]. This can also probably be applied for ServiceMix. Mule ESB has its own enterprise version, which is also subscription based [50]. Like Fusesource, Mulesoft too offers a large list of different options that the subscription gives. OpenESB's commercial support is more like ServiceMix's, having a few companies ready to give support [51]. Talend offers a feature-rich commercial version in addition to its community edition [47]. Petals did not have any information about any commercial support. WSO2 markets their support for their ESB solution to be from evaluation to production [52]. They have put quite a lot of effort into it. UltraESB offers three different levels of commercial support, with increasing amount of support and price [26].

4.3.9. IDE support

The tool support for current ESBs could use some improvements. Propriety ESBs may have a little better support for them than the open source ones. Eclipse and Netbeans can be used to develop integration solution for all the ESBs in the comparison group. However, having a good support for IDE, and thus increasing the developer productivity, is usually not given that much focus. Instead the focus is more on the runtime functionality and performance. Sometimes the effort to make the integration as easy as possible through and IDE can hurt the features of the ESB. For example, a drag and drop GUI might not help the productivity of the developers. [16]

Out of all the ESBs in the comparison group, OpenESB has the best IDE support with Netbeans. [16] This is one of the strongest points of OpenESB. Fuse has also its own IDE, Fuse IDE [53], which is based on Eclipse and can be used with Fuse or ServiceMix. Talend has plugins built on Eclipse, as does Mule. Petals has a Maven plugin and Petals Studio [54], a custom Eclipse distribution. WSO2 offers Carbon Studio [55], a supposedly fast IDE for SOA development. UltraESB has support for all the most popular IDEs, giving the choice of development tool to the developer [26]. The other ESBs have some IDE support, but these are not on the same level as OpenESB and UltraESB. This is something, that is hopefully improved in the future.

4.3.10. Previous experience

As a factor on choosing the right tool, previous experience is often not mentioned. Yet it is a major factor on which many developers base their final decision. When using a familiar tool, developers know exactly how to work with them and what they can or cannot do with them. If they have had good experience with a tool previously and think it will fit the requirements of a project, they will likely choose that. Then again, having bad experience with a tool previously can make a developer never use that tool again, even if the previous problems had been fixed. Still, having any experience with a tool previously aids a developer to be more productive.

In Valtimo, the developers have previous experience using OpenESB thus far. However, the experience has not been all that good. While there are some good aspects in it, there are also several problems with it, thus the need to change to a new one. Other than that, there is no previous experience for the rest of the ESBs.

4.3.11. Ease of use and usability

Usability is always an important factor when looking at any software. The tools should be as easy to use as possible, and the time to learn using them should be minimal. Easy to use and fast to learn software increase productivity and efficiency. Nielsen defines usability with five aspects [56]: learnability, efficiency, memorability, errors and satisfaction. Learnability means how easily user learns to use the product when using it for the first time. This could be also thought as how intuitive the product is. Efficiency is rather

self explanatory. It is measured by how quickly user can perform tasks with the product. Memorability means how well the users can use the product after a period of time away from it. Naturally, the less user has to memorize anything, the better. Errors is about how many errors user makes on average using the product. It is also important to note how severe those errors are and how easily users can recover from them. Lastly, satisfaction is about how pleasant the product is to use.

To fully determine the level of usability on each of the ESBs on the group, a deep usability research should be done. However, this would go beyond the extent of this thesis. Instead, the usability is measured by quick glance of the public consensus. All of the ESBs could use some usability improvements on many levels. ServiceMix is based on OSGi, which is not that intuitive at first glance. It is, however, rather efficient when user gets pass the learning stage. Fuse is there on the same level with ServiceMix. Fuse IDE might improve the user experience a bit. Mule has its own Studio tool, which could improve the usability. However, that tool is not free and without it users are stuck with the traditional way of creating integration routes. OpenESB has a nice integration with NetBeans, making the creation of solutions a bit easier. Overall, none of the ESBs really stick out with their usability.

4.3.12. Documentation

One criterion, which is sometimes overlooked, is how well a framework is documented. Well written documentation can be the deciding factor between two otherwise equal frameworks. It helps to set up the system, gives information about all the features of the framework and how to use them, and may help with any occurring problems.

The documentation should be easy to read and yet cover fully and accurately everything the user should know. It should contain only relevant information about the framework, and have easily understandable structure. Nielsen [57] says that software should be so easy to use that it does not need any documentation. However, if the software requires documentation, the required information should be easy and fast to find.

Mule has an excellet and easy to use documentation with a lot of examples and a nice layout [58]. ServiceMix's documentation seem to be a bit lacking [59]. Fuse has extended that by creating their own documentation, which gives a lot more information

and can be used with ServiceMix too [60]. Structure of the Fuse ESB's documentation could be better though. OpenESB offers a thorough documentation of the product and guides the processes step by step with pictures [61]. However, finding something exact may be slow, since the documentation is poorly structured. Talend does not make their documentation quite easily accessible as the rest, as it requires a registration in order to view it. That aside, they have a lot of documentation for the product and their designer studio. Petals has a documentation [23], which is fragmented on many different pages, and might be sometimes hard to navigate. However, it seems to have all the necessary information. WSO2 has clearly structured documentation section, which is easy to follow [62]. UltraESB has a similar layout [63].

4.3.13. Performance

Performance can become an issue if the ESB is put on extreme use. Usually all the ESBs can handle the message routing in a tolerable period of time, depending of the number of incoming messages of course. However, measuring the performance capabilities of an ESB is not that easy. A lot of vendors have their own performance tests, which rank their own ESB higher than the rest. There have been some efforts to make standardized SOA performance benchmarking, without much success. Some pitfalls of the current benchmarks are overemphasizing web services and thus SOAP (creating and parsing XML is costly), not focusing on transaction success rate or the affect of security protocols, and lastly, not taking concurrency into consideration. Any ESB can make them seem better than the rest by giving the benchmark some specific criteria. Without any standardization, however, no single benchmark can be truly trusted. [64]

Gathering data from multiple benchmarks, WSO2 and Ultra ESB seem to stand out with their performance. They have put some effort into gradually improving the performance of their products. The rest of the ESBs seem to be performing about equally well amongst each other.

4.4. Evaluation model

From the defined 13 criteria, the most important ones for Valtimo are: core functionality, open source, popularity, enterprise readiness, flexibility, licensing and connectivity.

Criteria	Weight	ServiceMix	Fuse	Mule	Open	Talend	Petals	WSO2	Ultra
Core functionality	2	+	+	+	+/-	+	+/-	+	+
Open source	2	++	+/-	+/-	++	++	++	++	+
Licensing	2	++	- -	++	++	++	++	++	- -
Popularity / Future	1.5	++	+/-	++	-	+	-	+	+/-
Enterprise readiness	1.5	++	+	++	-	+	-	+	+
Expandability / Flexibility	1.5	+	+/-	+	+/-	+	+/-	+	+
Connectivity	1	++	++	+	+/-	++	+	+	+
Total		19.5	1.5	14.5	5	16.5	6	15.5	4

Table 4.1: Evaluation of the ESBs against defined criteria.

The ESB should manage to do all the necessary routing, while also having a long life expectancy. It needs to be open source and have a license that does not restrict the development in any way. Finally, it should be flexible enough for the future needs.

To determine how different ESBs performed against each criteria, they shall be gathered together in a single table and weighed for how important each criteria is. From there, a final score can be calculated and a winner can be deduced. The results can be seen in Table 4.1. This table is specifically created to evaluate the suitability of an ESB for Valtimo, and cannot be used as a default list for choosing the best ESB.

The first column shows how important a criterion is to Valtimo. The rest of the columns show the points for each ESB. The points are given with pluses and minuses, ranging from two minuses to two pluses. They are measured on how well a criterion is met. Two pluses means excellent, one plus is good, plus slash minus indicates not standing out in any way, minus means poor, and two minuses really poorly. The total value is constructed by multiplying the weight with the current value and adding all the values together.

Some of the most important factors for this project are core functionality, open source and licensing. A bit less important, but still meaningful criteria include popularity, enterprise readiness and expandability. Then a step lower comes connectivity.

According to the table, the most suitable ESBs for Valtimo are currently ServiceMix, Mule, Talend, and WSO2. While Mule, Talend, and WSO2 are relatively similar to each other, ServiceMix seems to be the strongest choice for this situation. It will be used to create the integration solution.

5. IMPLEMENTING INTEGRATION SOLUTION WITH SERVICEMIX

In this chapter, implementing the integration solution will be fully described. This process consists of removing the current OpenESB solution from the system, and replacing it with Apache ServiceMix. Other parts of the system will remain the same. The Glassfish instances will also need to be configured accordingly. The chapter is divided into four parts: describing the integration architecture using EIPs, installing ServiceMix, changing the system to use message queues from ServiceMix instead of OpenESB, and finally, configuring the ServiceMix to process the messages correctly.

5.1. Integration architecture

To have an idea of the integration architecture of Valtimo, a model is made using the Enteprise Integration Patterns. This shows all the necessary queues, endpoints, transformers and translators required for the implementation. The model can be seen in Figure 5.1.

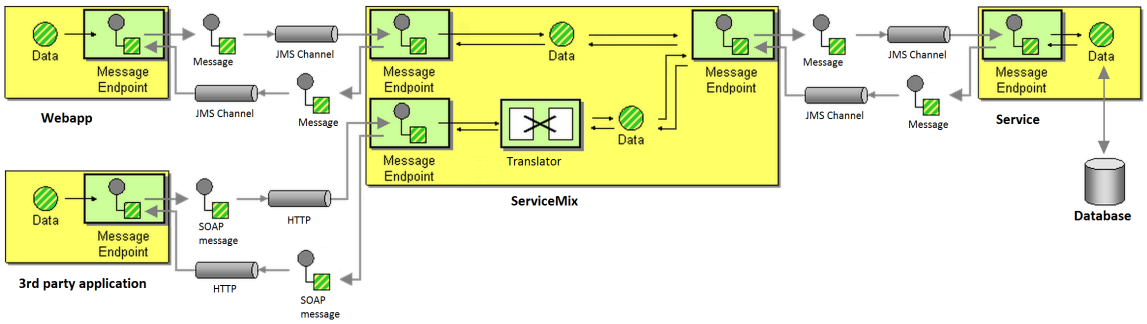

Figure 5.1: Integration architecture of Valtimo.

The figure shows four different components: webapp, service, ServiceMix and third party application. Webapp and service components communicate with regular JMS messages, while a third party application uses SOAP messages over HTTP. They will all relay the messages to ServiceMix, which needs to route them forward accordingly.

The system needs four queues: two between webapp and ServiceMix, and two between ServiceMix and service component. These are for the Request Reply pattern, where one queue is for the request messages, and one for the responses. Only the request queues need to be static, since the return messages can be made with temporary queues. ActiveMQ documentation recommends to create a single temporary queue and consumer for each client on startup [65].

The WSDL interface is built so that ServiceMix is listening for SOAP messages over HTTP and sending them to the service component. The messages come from a third party application, which needs to send valid SOAP messages so that ServiceMix can handle them. ServiceMix cannot form the messages correctly automatically, so a translator is needed to have correct parts in headers and in body. The return message needs to be translated too.

There is a possibility to change this architecture a little by creating additional queues for load balancing. Some of the requests are rather heavy and may cause performance issues for other users. To prevent this, a content based router could be added, which routes the heavier requests to another queue, leaving the other queue for all the rest. However, this routing is not necessary if the performance will not be an issue to the users.

5.2. Installing ServiceMix

Installing ServiceMix is quite straightforward. It does not have any system requirements, apart from a working Java Runtime Environment (JRE) 6 or 7, and Java Development Kit (JDK) 6 or 7 [66]. Also, for developing integration applications, Apache Maven [67] (3.0.4 or higher) is required. Currently project Valtimo uses Java 7 with Maven 3.0.2 or higher [28]. For this system, Apache ServiceMix 4.5.3 [68] was downloaded. It works under the Apache License v2. [69]

After installing ServiceMix (by simply uncompressing the package), it can be started from a command prompt with a *servicemix* command. This starts the Karaf OSGi runtime with the ServiceMix on top of it. Example can be seen in Figure 5.2. Within the console, features of the ServiceMix can be managed. [69]

By using the command *osgi:list*, all the currently installed bundles can be seen [69]. By default, there are several bundles already installed, as seen in Figure 5.3. The most

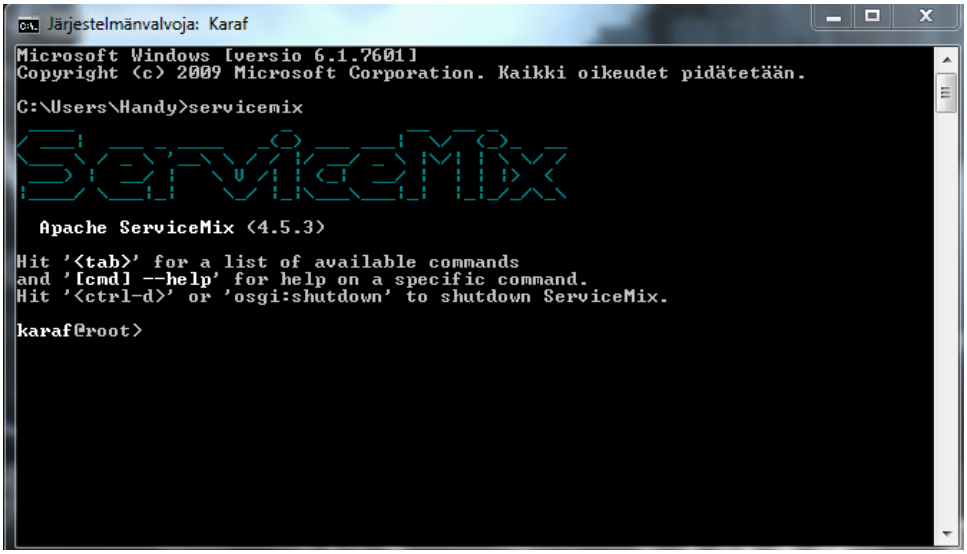

Figure 5.2: Starting up ServiceMix.

notable ones are Camel, ActiveMQ, and CXF related bundles, which will be needed in the solution. The list also gives some information about the bundles. Their state can be installed, resolved (all the dependencies are satisfied, bundle is stopped or ready to start), starting, active, stopping or uninstalled [70]. If the bundle contains a Blueprint or Spring XML file, their states will be shown in their respective columns. Blueprint provides a framework for dependency injection for OSGi, and is designed to take care of the dynamic nature of it. Bundles can become available or unavailable at any time. Also, it is possible to write simple components using POJOs (Plain Old Java Objects). Blueprint XML files define how various components are instantiated and wired together [71]. Different states for blueprint and Spring XML fields include created, creating, destroyed, failure (exception or a missing dependency), grace period (unsatisfied dependency), and waiting. The last two fields in the list are start level, which defines the start order of the bundles, and name, which has the name of the bundle and the version of it. If the start level of a bundle is higher than the start level of the system (active start level), the bundle will not be started at all [70].

Now ServiceMix is ready to be used. To check that it started without problems, a log can be seen with the *log:display* command. There is also a web console available for ServiceMix, which can be used to configure and manage it. By default, it runs on port 8181. To manage ActiveMQ, a separate web console needs to be installed. It can be installed by first installing war features with *features:install war* and then the web console with *fea-*

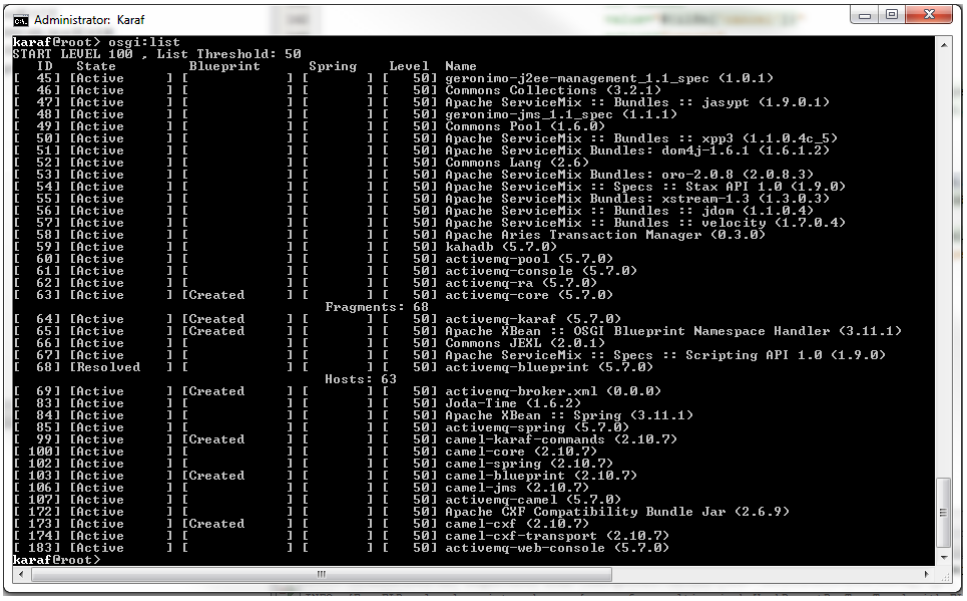

Figure 5.3: ServiceMix feature list.

tures:intall activemq-web-console. After this, the console can be accessed by going to the following address: `http://localhost:8181/activemqweb/index.jsp`. The default port can be changed by creating a org.ops4j.paw.web.cfg configuration file in the ServiceMix's etc folder and specifying org.osgi.service.http.port attribute. The greeting screen of ActiveMQ web console is shown in Figure 5.4. [69]

Figure 5.4: ActiveMQ web console.

5.3. Replacing OpenMQ with ActiveMQ

The first step of changing the system to use ServiceMix instead of OpenESB is to change the messages to use ActiveMQ provided by ServiceMix instead of the OpenESBs OpenMQ. This is done, because every ServiceMix instance comes with an ActiveMQ broker embedded [69]. Thus, it is easier to use ActiveMQ than any other broker. The replacement is done mainly through configuring the Glassfish instances. The configuration

is done by using the Glassfish admin console. [72]

The Glassfish server has a few different parts that need to be configured. The queues are managed with JMS connection factories and destination resources. Connection factories are for making connections to the queues, and destination resources are references to queues [29]. By default, Glassfish does not support using queues created with ActiveMQ. For this, a specific resource adapter needs to be used, which Apache provides in their website [73]. After downloading the adapter, it can be deployed in the Glassfish by using the "Deploy Application or Modules" feature. Also, a few libraries from ActiveMQ needs to be added to Glassfish (slf4j-api-1.5.11, slf4j-log4j12-1.5.11 and log4j-1.2.14). The default configuration of the resource adapter can be seen in Figure 5.5. The ServerUrl attribute is the address from which the destination resources will be looking for the physical queues. In this case, it should be address for the ActiveMQ service in ServiceMix. [72]

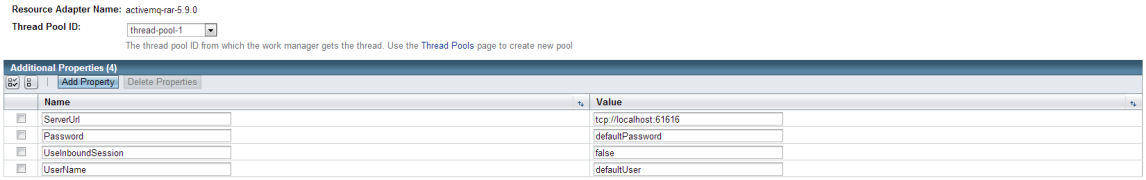

Figure 5.5: Resource adapter configuration.

Next, the OpenMQ queues need to be changed to ActiveMQ queues. This can be done by changing the Resource Adapter of the queues from the default jmsra to the newly created ActiveMQ adapter and changing the "Name" attribute to "PhysicalName" [72]. However, Glassfish admin console seems to have some problems creating these queues properly. It may be required to create the queues from the console by starting asadmin and using the "create-admin-object" command. Otherwise the application may give an error about JNDI lookup failing. In other words, it does not register the queues correctly. Example queue can be seen in Figure 5.6.

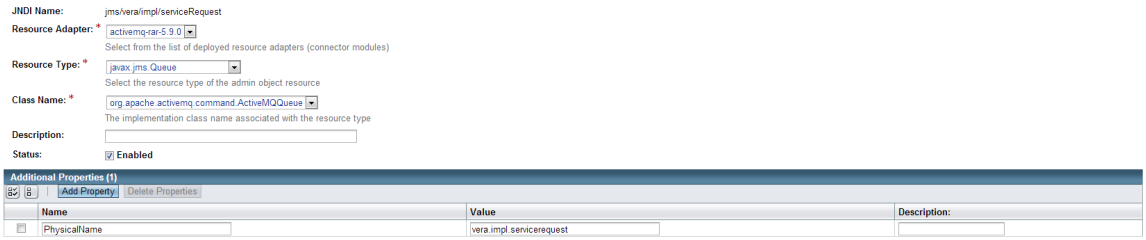

Figure 5.6: Resource adapter configuration.

After the queues are set, Glassfish still needs connection factory to be set up. This can

be done from the admin console by creating a connection pool, that uses the ActiveMQ resource adapter, and changing the current connection factories to use it. Every connection factory needs a connection pool to handle the connections [29]. The pool settings are left as default in this solution. [72]

Now Glassfish should be ready to use queues from ActiveMQ. The only thing left to do is to configure our current application to take these changes into account. In the current system, web application sends messages to queues using JAXB (Java Architecture for XML Binding). The messages need to be marshalled before sending, and the reply needs to be unmarshalled to make the data formats compatible. Sending the message consists of making a connection with a connection factory (which is referenced with its JNDI name), creating a session from the connection, and using the JMS's MessageProducer and TextMessage classes to form the message and send it to a queue (which is also referenced by its JNDI name). After this, the application waits for a reply message to a predetermined queue. When the process is finished, both the session and the connection are closed. This system will not need any changes, since the process and JNDI names are still the same. [29]

The service side, however, will need some fixing. In it, there is a message driven bean, which is supposed to listen to any messages coming from a queue with a specific JNDI name. It uses a @MessageDriven annotation, which determines this information within its attributes. Unlike OpenMQ, however, ActiveMQ will not need the "mappedName" attribute, but instead a "destination" property, and unlike mapped name, this references the actual queue name instead of the JNDI name. Example is shown in Program 5.1. Also, to let the message driven bean to known that we are using ActiveMQ, sun-ejb-jar.xml should have a resource adapter attribute like in Program 5.2. Otherwise the message bean has the same principles as in the previous case. It receives a message in the message listening function, processes its content (does what the message asks it to do), creates a connection and sends a response.

Program 5.1: Message driven bean annotation

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.
        jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination", propertyValue = "vera.impl.
        servicerequest")
```

```
})
```

Program 5.2: Message driven bean resource adapter

```
<mdb-resource-adapter>
   <resource-adapter-mid>activemq-rar -5.9.0</resource-adapter-mid>
</mdb-resource-adapter>
```

5.4. Configuring ServiceMix

Now that the queues are set up, it is time to put them through ServiceMix instead of OpenESB. In order to create the necessary routing, several steps need to be made. These were found out partly by the ServiceMix installation guide [69], partly by Camel documentation [74], and partly by just trial and error. First of all, since ServiceMix runs on Karaf, it requires a proper OSGi bundle to handle the routing. These can be made with any IDE, but in this case we will be making one with Netbeans. Once the bundle is made, it can be deployed into ServiceMix by copying it into the deploy folder [69]. After that, ServiceMix will instantly recognize the new bundle and try to install it and put it running. If the bundle is deployed successfully, the routing should be working. It and its status can be checked with the *osgi:list* command.

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
    <version>2.10.7</version>
    <type>jar</type>
</dependency>
```

Figure 5.7: POM camel dependency.

To create an OSGi bundle in Netbeans, a new Maven "OSGi Bundle" project is made. After giving it a name and all the necessary information, it will create a project and some of the files needed for the bundle. The first thing to do is configure the POM (Project Object Model). POM includes all the configurations for a project and handles any dependencies through Maven [75]. The automatically generated POM will be completely rewritten with our own code, except for the basic information about the project name, version, etc. For dependencies, we will need the Camel core from `org.apache.camel` as seen in Figure 5.7 [69]. The version used is the same that is installed in ServiceMix. For the plugins, there are two requirements. One is for constructing a Maven bundle, and

the other one is for Apache CXF, which is required for the Web Services interface. We will get to the CXF later. The Maven plugin is shown in Figure 5.8. It is an instruction for Maven on how to build the bundle. The private package tells where to find our custom packages, which include the routing.

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <version>2.3.7</version>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
            <Import-Package>
                *,
                org.apache.camel.osgi
            </Import-Package>
            <Private-Package>com.gofore.valtimo.integration</Private-Package>
        </instructions>
    </configuration>
</plugin>
```

Figure 5.8: POM Maven Bundle plugin.

After that, we need to create a blueprint so that ServiceMix understands to register our new routing with the system [69]. It is an XML file, which defines a blueprint and some Camel context. We will create it under resources/OSGI-INF folder. The contents of the file are rather simple, as seen in Figure 5.9. Camel context only defines a package for our routing.

```
<blueprint
    xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:cxf="http://camel.apache.org/schema/blueprint/cxf"
    xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
                        http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">
    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <!-- install the Java DSL route builder -->
        <package>com.gofore.valtimo.integration</package>
    </camelContext>
</blueprint>
```

Figure 5.9: Blueprint XML file.

To actually implement the routing, we will create a route builder class. More specifically, we will create a Java class which will extend the Camel's RouteBuilder class [74].

Inside, we will make any Camel routing necessary with the Java DSL. In this case, two routings are required. The first one is for the messages coming from our web application. The message should be forwarded to the service component. The second route is for the response message from service component, which in turn should be forwarded back to the web application. The two routes can be seen in Figure 5.10. The DSL is rather simple, telling messages from one queue to be sent to another, while also logging us the message body.

```
from("activemq:vera.servicerequest")
        .log("Received msg: ${body}")
        .to("activemq:vera.impl.servicerequest");

from("activemq:vera.impl.serviceresponse")
        .log("Received msg: ${body}")
        .to("activemq:vera.serviceresponse");
```

Figure 5.10: Routing with the Camel DSL.

Now the JMS messags should be working properly, and to test this we can deploy the bundle into ServiceMix and test the application. Listing all the osgi bundles we should see our newly deployed bundle having been created as in Figure 5.11. Testing the application, it seems like the routes are working fine and messages are going through correctly.

```
[ 213] [Active     ] [Created     ] [       ] [   80] ValtimoIntegration (1.0.0.SNAPSHOT)
```

Figure 5.11: Created bundle in ServiceMix.

The only thing left to do after this is to configure the Web Service interface. This is done with CXF, and the Web Service can be implemented in the same bundle as the JMS routing. We already have some of the WSDL files, that have been used in the previous solution. They define the message format. What is still needed, is the service definition and a binding for the port element it uses [7]. These are quite straightforward. The port element needs an address where the service will be published to, and the binding requires operation definitions, which tell what we can do with the interface. In this case, we will create a "sendRequest" operation, which will have a "request" input element and a "response" output element.

After the WSDL files are set, we need to create Java files from them to complete the definition. This is done with Maven, using the plugin mentioned before [74]. It requires the main WSDL file as a parameter and it will use CXF's wsdl2java tool for the

generation. The code is shown in Figure 5.12.

```
<plugin>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-codegen-plugin</artifactId>
    <version>2.6.9</version>
    <executions>
        <execution>
            <id>generate-sources</id>
            <phase>generate-sources</phase>
            <configuration>
                <sourceRoot>${project.build.directory}/generated-sources/cxf</sourceRoot>
                <wsdlOptions>
                    <wsdlOption>
                        <wsdl>${basedir}/src/main/resources/META-INF/wsdl/ServiceRouter.wsdl</wsdl>
                    </wsdlOption>
                </wsdlOptions>
            </configuration>
            <goals>
                <goal>wsdl2java</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Figure 5.12: POM CXF code generation plugin.

To actually register the created Web Service interface into ServiceMix, we will add a CXF endpoint into the blueprint file [69]. It will need an address where it will publish the service and a service class which includes the actual service. The service class is one of the generated classes that the Maven plugin created. Additionally, we will add a data format property for the endpoint. In this case we will set it to "MESSAGE", which will pass the raw message received from the transport layer, without doing anything to it [74]. The endpoint is seen in Figure 5.13.

```
<cxf:cxfEndpoint id="valtimo-proxy"
                 address="/vera"
                 serviceClass="fi.stm.vera.wsdl.services.request.ServiceRequestPortType">
    <cxf:properties>
        <entry key="dataFormat" value="MESSAGE" />
    </cxf:properties>
</cxf:cxfEndpoint>
```

Figure 5.13: CXF endpoint definition in Blueprint.

Now the final thing to do is to set up routing for the messages coming through the CXF endpoint. The route is identified with the id we gave to the CXF endpoint in the blueprint file. Routing the message is a little trickier than the JMS routes, since we cannot send the messages through as they are. The message, which is XML, needs to be cut in two. The SOAP header goes to the "UsernameToken" in JMS header, and the SOAP body will be set as the new JMS body. This can be done with xpath, which is a query language for selecting nodes in XML document. We will use it to select the "UsernameToken" and

"InvokeService" elements and putting them to their right places. However, these elements have their own namespaces, so in order for xpath to work, we will need to define these namespaces and use them with xpath. Also, since the message comes in a stream, it can be read only once in Camel. To fix this, Camel has a stream caching option, which allows to read the content multiple times. Finally, the transformed message is sent to a JMS queue, that the service component is listening. The final route definition is shown in Figure 5.14. It has a few logs to ensure that the message is transformed correctly and a correct response is received. These will be removed later.

```
from("cxf:bean:valtimo-proxy").routeId("vera")
        .streamCaching()
        .log("Received msg: ${body}")
        .setHeader("UsernameToken", ns.xpath("//c:Header/h:UsernameToken", String.class))
        .setBody(ns.xpath("//c:Body/b:InvokeService"))
        .log("Header Token: ${header.UsernameToken}")
        .log("Transformed msg: ${body}")
        .to("activemq:vera.impl.servicerequest")
        .log("Received msg response: ${body}");
```

Figure 5.14: CXF routing.

To test the Web Service implementation, the updated bundle will be deployed in to ServiceMix. If any services are found, they will be shown in `http://www.localhost:(DefaultPortForServiceMix)/cxf`. In this case, we seem to have one service published, as seen in Figure 5.15. We can use this to verify the endpoint address, which we will need for our Web Service clients. It will also tell us the service, which we can call.

Available SOAP services:

serviceRequestPortType • sendRequest	Endpoint address: http://localhost:8282/cxf/vera WSDL : {http://vera.stm.fi/wsdl/services/request}ServiceRequestPortTypeService Target namespace: http://vera.stm.fi/wsdl/services/request

Figure 5.15: Available SOAP services.

The final step is to generate a Web Service client, and create a simple program around it to call the service and print us the result. The client can be generated with the same Maven plugin, that was used earlier to make the server side Java files for ServiceMix. After making sure that the Web Service interface is working, the implementation is complete.

Looking at the used Enterprise Integration Patterns in the new solution, it has rather many utilized. The core is built up with Messaging, and some of the patterns revolving around it. These include Message Channel, Message, Message Translator, and Message

Endpoint. Also, like every other ESB, the solution uses Message Bus to create a loosely coupled messaging system. On a more specific level, the new solution uses Point-to-Point Channels, which transfer Command Messages with a Request-Reply pattern. The messages use Return Address to determine where the reply will be sent, and Correlation Identifier to know which request the reply is for. Also, the messages use Message Expiration pattern to ensure the messages won't be waited for forever. For the CXF messaging, a Content Enricher is used to add certain needed headers to the messages. Lastly, the solution uses Messaging Gateways to encapsulate the messaging related code from the rest of the application.

Implementing a certain pattern was not hard, since Camel includes documentation on how to implement each pattern [76]. This also makes it easier to implement some more features in the future. For example, a content based message router to ease load balancing should be rather easy to implement with the help of the documentation. Since more patterns are now available, the new solution is also more flexible than the previous one.

6. CONCLUSION

Enterprise Integration Patterns can help the development of integration solutions by giving a set of best practices to create the architecture. It also encourages integration frameworks to support the patterns, and thus brings a minimum requirement level for the features that a framework must provide. In Valtimo, they helped to create the layout for the integration. For the actual implementation, built with ServiceMix ESB, they were involved through Camel, which had references to the patterns in the documentation and includes instructions on how to implement them with it.

However, having a framework that is based on EIPs does not automatically mean it is optimally made. A lot is dependant on the implementation of the framework too and how big of an audience it has gotten. The EIPs only give a base on which to build on.

Choosing the right tool for a specific integration need is very important. Integration can happen on many different levels, so the tools should be chosen accordingly. The most simple cases can be done with point-to-point integration. Integration frameworks, such as Camel or Spring Integration, can be used to help the development. However, if the solution is a bit more complicated, or needs multiple different clients or connectors, an ESB can help to organize the solution and keeping the different component as loosely coupled as possible. In Valtimo's case, using an ESB seemed like the most logical choice, since the architecture is SOA focused and the previous solution was ESB based too. To find the best ESB to use, a certain set of criteria was created based on how important they were to Valtimo. After evaluating different ESBs against the criteria, the most suitable ESB, ServiceMix, was found.

Comparing the new solution to the old one, which built with OpenESB, there are some noticeable differences. Camel brings the benefits of EIPs, which in this case are standardized ways to create an integration solution, and providing a richer set of features. Camel seems to also have an active community behind it, so it probably has a longer life span and will be updated more frequently. ServiceMix is being kept up to date whenever the

components in it (Camel, ActiveMQ, Karaf, CXF, etc.) are updated. Camel is also more flexible by having so many different connectors in it and enabling developers to easily write any custom logic to it. Performance wise there are no noteworthy differences.

The current solution allows developers to easily add any features in the future. If the performance needs optimizing, additional prioritized queues can be added for that. That way important messages will go to a different queue and will more likely have a faster response. If the solution needs more connectors or transformers, they can be easily added to the ServiceMix router. Deploying the new solution can be done in just a few steps. The current solution is also expected to be flexible enough to be adapted to any future requirements.

BIBLIOGRAPHY

[1] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. Pearson Education, Inc., Boston, MA, USA, 2012.

[2] Gregor Hohpe. Integration Patterns Overview. `http://www.eaipatterns.com/eaipatterns.html`, 2012. Referenced: 3.11.2013.

[3] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service-oriented Architecture Best Practices*. Pearson Education, Inc., Upper Saddle River, NJ, USA, 2004.

[4] Thomas Erl. *SOA: Principles of Service Design*. Prentise Hall, Boston, MA, USA, 2008.

[5] Thomas Erl. SOA Manifesto. `http://www.soa-manifesto.org`, 2013. Referenced: 27.10.2013.

[6] Rick Sweeney. *Achieving Service Oriented Architecture*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2010.

[7] W3. Web Services architecture. `http://www.w3.org/TR/ws-arch`, 2004. Referenced: 24.11.2013.

[8] Swati Dhingra. REST vs. SOAP: How to choose the best Web Service. `http://searchsoa.techtarget.com/tip/REST-vs-SOAP-How-to-choose-the-best-Web-service`, 2013. Referenced: 21.03.2014.

[9] Wikipedia. Web Service. `http://en.wikipedia.org/wiki/Web_service`, 2013. Referenced: 24.11.2013.

[10] Gian Trotta. Dancing Around EAI 'Bear Traps'. `http://www.ebizq.net/topics/int_sbp/features/3463.html`, 2003. Referenced: 3.11.2013.

[11] Christopher Alexander. *A Pattern Language*. Oxford University Press, University Offices, Wellington Square, Oxford, 1977.

[12] Kai Wähner. When to use Apache Camel. http://www.kai-waehner.de/blog/2011/06/02/when-to-use-apache-camel, 2011. Referenced: 16.11.2013.

[13] Jonathan Anstey. Open Source Integration with Apache Camel and How Fuse IDE can Help. http://java.dzone.com/articles/open-source-integration-apache, 2011. Referenced: 16.11.2013.

[14] SpringSource. Spring Integration. http://projects.spring.io/spring-integration, 2013. Referenced: 17.11.2013.

[15] Kai Wähner. Which Integration Framework Should You Use - Spring Integration, Mule ESB or Apache Camel? http://java.dzone.com/articles/which-integration-framework, 2012. Referenced: 17.11.2013.

[16] Tijs Rademakers and Jos Dirksen. *Open Source ESBs In Action*. Manning Publications Co., Sound View Court 3B, GreenWich, CT 06830, 2009.

[17] Gregor Hohpe. Hub and Spoke Zen and the Art of Message Broker Maintenance. http://www.eaipatterns.com/ramblings/03_hubandspoke.html, 2013. Referenced: 18.11.2013.

[18] Loek Bakker. Goodbye Hub-and-Spoke, Hello ESB? Integration Architecture With BizTalk 2004. http://dotnet.sys-con.com/node/121831, 2005. Referenced: 18.11.2013.

[19] Ross Mason. OSGi? No Thanks. http://blogs.mulesoft.org/osgi-no-thanks, 2010. Referenced: 22.11.2013.

[20] Guillaume Nodet. Thoughts about ServiceMix. http://gnodet.blogspot.fi/2010/12/thoughts-about-servicemix.html, 2010. Referenced: 10.11.2013.

[21] OpenESB. http://www.open-esb.net/, 2014. Referenced: 03.05.2014.

[22] Kai Wähner. Choosing the right ESB for your integration needs. http://www.infoq.com/articles/ESB-Integration, 2013. Referenced: 10.01.2014.

[23] Petals ESB. http://petals.ow2.org/, 2014. Referenced: 23.03.2014.

[24] ESB Performance. http://esbperformance.org/display/comparison/ESB+Performance, 2014. Referenced: 15.05.2014.

[25] WSO2 ESB. http://wso2.com/products/enterprise-service-bus/, 2014. Referenced: 23.03.2014.

[26] UltraESB. http://www.adroitlogic.org/products/ultraesb.html, 2014. Referenced: 31.03.2014.

[27] STM - Työsuojelun tarkastusjärjestelmä. http://gofore.com/asiakkaat/#stm-tyosuojelun-tarkastusjarjestelma, 2014. Referenced: 04.05.2014.

[28] Valtimo - Järjestelmän arkkitehtuurin kuvaus. https://extra.gofore.com/confluence/pages/viewpage.action?pageId=6096621 (internal), 2014. Referenced: 04.05.2014.

[29] Valtimo - Viestinvälitys. https://extra.gofore.com/confluence/pages/viewpage.action?pageId=6097026 (internal), 2014. Referenced: 04.05.2014.

[30] Peter Walker. JSR 312: Java Business Integration 2.0. http://jcp.org/en/jsr/detail?id=312, 2010. Referenced: 10.11.2013.

[31] Ross Mason. To ESB or not to ESB. http://blogs.mulesoft.org/to-esb-or-not-to-esb/, 2009. Referenced: 07.02.2014.

[32] Tapio Rautonen. Kehysten käyttö Java EE-ohjelmistoprojekteissa, 2009. Tampere University of Technology.

[33] EIP Patterns using BPEL. http://predic8.com/bpel-xpath-splitter.htm, 2009. Referenced: 05.05.2014.

[34] Working with EIP in Petals. https://doc.petalslink.com/display/petalsstudiosnapshot/Working+with+EIP+in+Petals, 2014. Referenced: 05.05.2014.

[35] Enterprise Integration Patterns with WSO2 ESB. https://docs.wso2.org/display/IntegrationPatterns/Enterprise+Integration+Patterns+with+WSO2+ESB, 2014. Referenced: 05.05.2014.

[36] Enterprise Integration Patterns. http://docs.adroitlogic.org/display/esb/Enterprise+Integration+Patterns, 2014. Referenced: 05.05.2014.

[37] Fuse Developer Program Terms and Conditions. http://www.jboss.org/developer-program/termsandconditions, 2014. Referenced: 05.05.2014.

[38] Apache License, Version 2.0. http://www.apache.org/licenses/LICENSE-2.0.html, 2014. Referenced: 05.05.2014.

[39] Common Public Attribution License Version 1.0. http://opensource.org/licenses/CPAL-1.0, 2014. Referenced: 05.05.2014.

[40] Common Development and Distribution License. http://opensource.org/licenses/CDDL-1.0, 2014. Referenced: 05.05.2014.

[41] GNU Lesser General Public License. https://www.gnu.org/licenses/lgpl.html, 2014. Referenced: 05.05.2014.

[42] GNU Affero General Public License. http://www.gnu.org/licenses/agpl-3.0.html, 2014. Referenced: 05.05.2014.

[43] Mule ESB. https://www.mulesoft.com/platform/soa/mule-esb-open-source-esb, 2014. Referenced: 06.05.2014.

[44] Alex Glazkov. Is OpenESB dead? http://aglazkov.wordpress.com/2010/05/18/is-openesb-dead/, 2010. Referenced: 06.05.2014.

[45] Writing a Mediator in WSO2 ESB. http://wso2.com/library/2898/, 2014. Referenced: 06.05.2014.

[46] UltraESB Mediation Reference. http://docs.adroitlogic.org/display/esb/Mediation+Reference, 2014. Referenced: 06.05.2014.

[47] Talend ESB. http://www.talend.com/products/esb, 2014. Referenced: 06.05.2014.

[48] ServiceMix Support. http://servicemix.apache.org/community/support.html, 2014. Referenced: 08.05.2014.

[49] Fuse Enterprise Subscription. http://fusesource.com/collateral/download/53, 2014. Referenced: 08.05.2014.

[50] Mule ESB Enterprise. http://www.mulesoft.com/platform/soa/mule-esb-enterprise, 2014. Referenced: 08.05.2014.

[51] Open ESB Support. http://www.open-esb.net/index.php?option=com_content&view=article&id=81&Itemid=500, 2014. Referenced: 08.05.2014.

[52] WSO2 Support. http://wso2.com/support/111, 2014. Referenced: 08.05.2014.

[53] Fuse IDE. https://access.redhat.com/site/documentation/en-US/Fuse_IDE/, 2014. Referenced: 08.05.2014.

[54] Petals Studio. http://www.petalslink.com/en/products/petals-studio, 2014. Referenced: 08.05.2014.

[55] WSO2 Developer Studio. http://wso2.com/products/developer-studio/, 2014. Referenced: 08.05.2014.

[56] Jacob Nielsen. Usability 101. http://www.nngroup.com/articles/usability-101-introduction-to-usability/, 2012. Referenced: 03.03.2014.

[57] Jacob Nielsen. 10 Usability Heuristics for User Interface Design. http://www.nngroup.com/articles/ten-usability-heuristics/, 1995. Referenced: 20.01.2014.

[58] Mule ESB Documentation. http://www.mulesoft.org/documentation/display/current/Home, 2014. Referenced: 08.05.2014.

[59] ServiceMix Documentation. http://servicemix.apache.org/docs/5.0.x/index.html, 2014. Referenced: 08.05.2014.

[60] Fuse ESB Documentation. http://fusesource.com/documentation/fuse-esb-documentation/, 2014. Referenced: 08.05.2014.

[61] OpenESB Documentation. http://www.open-esb.net/index.php?option=com_content&view=article&id=86:openesb-documentation&catid=80:openesb-documentation&Itemid=488, 2014. Referenced: 08.05.2014.

[62] WSO2 ESB Documentation. https://docs.wso2.org/display/ESB460/Enterprise+Service+Bus+Documentation, 2014. Referenced: 08.05.2014.

[63] UltraESB Documentation. http://docs.adroitlogic.org/display/esb/AdroitLogic+UltraESB+-+Documentation, 2014. Referenced: 08.05.2014.

[64] Martin Vecera. ESB Performance Pitfalls. http://soa.dzone.com/articles/esb-performance-pitfalls, 2010. Referenced: 09.03.2014.

[65] How should I implement request response with JMS? http://activemq.apache.org/how-should-i-implement-request-response-with-jms.html, 2014. Referenced: 20.04.2014.

[66] Java SE. http://www.oracle.com/us/technologies/java/standard-edition/overview/index.html, 2014. Referenced: 08.05.2014.

[67] Maven. http://maven.apache.org/, 2014. Referenced: 08.05.2014.

[68] ServiceMix. http://servicemix.apache.org/, 2014. Referenced: 08.05.2014.

[69] ServiceMix Installation. http://servicemix.apache.org/docs/5.0.x/quickstart/installation.html, 2014. Referenced: 08.05.2014.

[70] OSGi Bundle. http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/Bundle.html, 2014. Referenced: 09.05.2014.

[71] Apache. Blueprint. http://aries.apache.org/modules/blueprint.html, 2013. Referenced: 30.11.2013.

[72] How to connect Glassfish 3 to an external ActiveMQ 5 broker. http://geertschuring.wordpress.com/2012/04/20/how-to-connect-glassfish-3-to-activemq-5/, 2014. Referenced: 09.05.2014.

[73] ActiveMQ Resource Adapter. http://activemq.apache.org/resource-adapter.html, 2014. Referenced: 09.05.2014.

[74] Camel Documentation. http://camel.apache.org/documentation.html, 2014. Referenced: 09.05.2014.

[75] Introduction to POM. http://maven.apache.org/guides/introduction/introduction-to-the-pom.html, 2014. Referenced: 09.05.2014.

[76] Camel - Enterprise Integration Patterns. http://camel.apache.org/eip.html, 2014. Referenced: 10.05.2014.